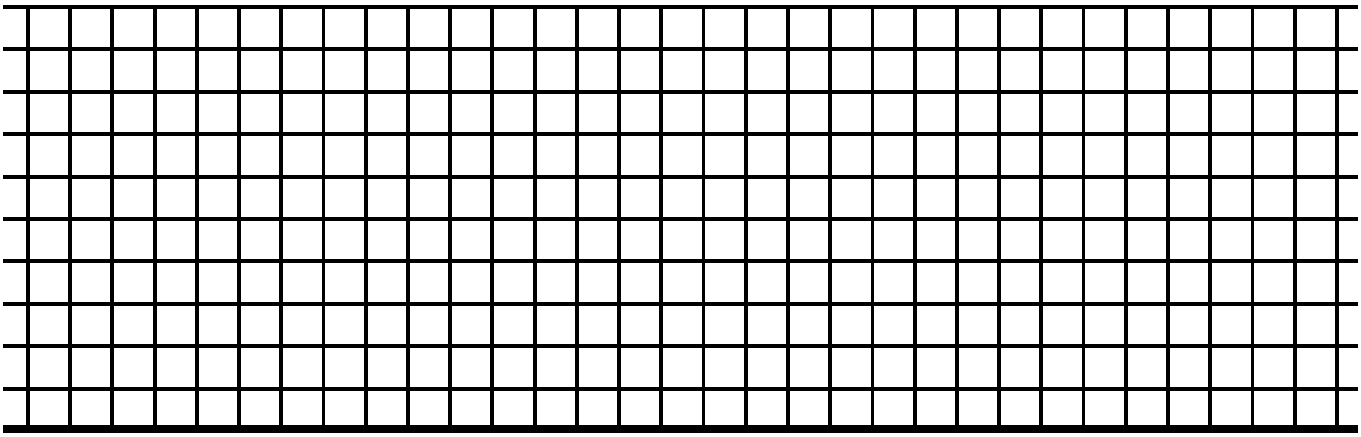
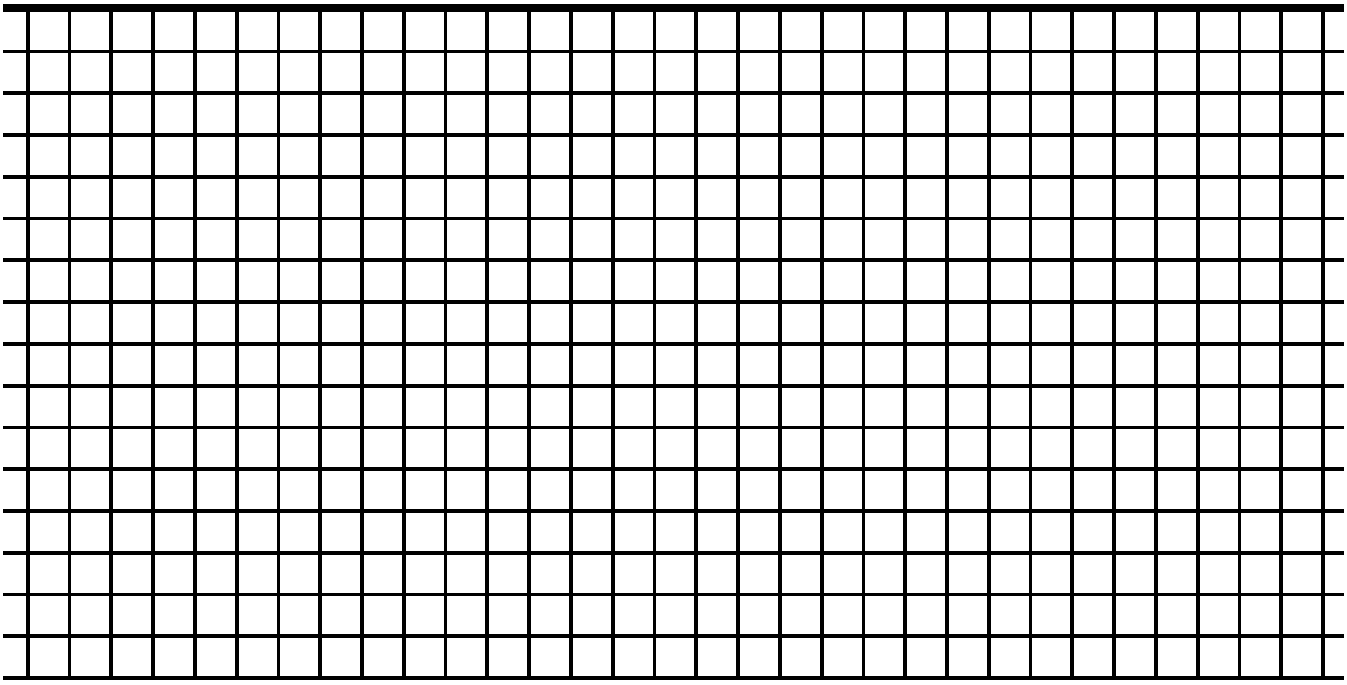


LADISLAV ZAJÍČEK

BITY DO BYTU

Základy programování ve strojovém kódu - assembleru Z80

MLADÁ FRONTA / PRAHA 1988



© Ladislav Zajíček, 1988
Illustration © Jaroslav Baierle, 1988

OBSAH

O ČEM JE TAHLE KNÍŽKA
Předmluva lektora **(5)**
Předmluva autora **(7)**

1 část
Týden v předsálí softwarového paláce

Bitové pondělí s byty a bajty **(11)**
Úterý zaslíbené paměti **(20)**
Hexadekadická středa **(25)**
Logický čtvrtek pana Boolea **(29)**
Zakódovaný pátek **(31)**
Mikroprocesorová sobota **(35)**
Neděle na lince **(38)**

2 část
Instrukční soubor mikroprocesoru Z80

KAPITOLA 1 Adresovací mody Z80 — 1 část **(45)**
KAPITOLA 2 Adresovací módy Z80 - 2 část (**62)**
KAPITOLA 3 Skoky volání a návraty **(79)**
KAPITOLA 4 Logické instrukce **(95**)
KAPITOLA 5 Bitové manipulace: rotace a posuvy **(104)**
KAPITOLA 6 Aritmetické instrukce a blokové prohledávaní **(116)**
KAPITOLA 7 O interfacingu **(129)**

3. část
Den po **(145)**

O ČEM JE TAHLE KNÍŽKA

PŘEDMLUVA LEKTORA

Podle hrubého tříděni se dá svazek, který držíte v ruce zařadit mezi naučnou literaturu. Čtenář naučné literatury obvykle přistupuje k pultu knihkupectví s poměrně jasnou představou o knížce, kterou si chce koupit. Neví třeba, jestli bude dobrá nebo špatná, skutečně zasvěcena nebo povrchní, čtivá nebo nudná, v každém případě však ví o čem ta knížka je. Kupuje si ji přece právě proto, aby se cosi dozvěděl o předmětu nebo oboru, který ho zajímá.

Přesto si troufnu předpokládat, že čerstvý majitel této knížky vlastně příliš neví, o čem to bude. Tedy až na výjimky, o nichž se zmíním později. Teď mám na myslí pravě toho čtenáře, kterému je knížka určena. Je pravděpodobně majitelem mikropočítače vybaveného mikroprocesorem Z80 nebo má alespoň k němu přistup. Má asi za sebou obvyklý začátek amatérského programování, to jest ovládá Basic a má velkou ctižádost dokázat s počítačem víc než chudý a pomalý Basic umožňuje. Hádám, že už si také aspoň trochu zakoketoval se strojovým kódem — přinejmenším tím, že použil nějakou hotovou rutinu, jichž je zejména mezi sinclairisty spousta v oběhu. Ví tedy zhruba tolik, že strojový kód je onen jazyk, kterým si počítač povídá sám se sebou, ví, že existuje polidštěna forma strojového kódu zvaná assembler, ví samozřejmě, že přes stroják, respektive assembler vede cesta ke skutečně dokonalému ovládnutí počítače. Což je pravě to po čem touží, aby ho počítač opravdu na slovo poslouchal

Ví tedy co chce, čeho by chtěl dosáhnout a přibližně tuší, že to nebude docela snadné. O vlastním obsahu předmětu, kterým se hodlá zabývat však zřejmě neví téměř nic. Kdybych chtěl být škodolibý doložil bych toto své tvrzení argumentem, že kdyby měl tušeni do čeho se pouští, nepouštěl by se do toho

Nebudu dál tajit co jsem vlastně už dost zřetelně naznačil, hovořím tak zasvěceně o situaci laického zájemce o programování v assembleru, protože ji dobře znám z vlastni zkušenosti. Před nemnoha měsíci jsem byl na stejném břehu, na kterém stojí čtenář, jemuž je tato kniha určena - a stejně jako on jsem trochu váhal zda do té vody doopravdy skočím. Ve srovnání s ním jsem byl v nevýhodě, učil jsem se ze špatné knížky. Nicméně skočil jsem. A pak jsem nějaký čas pouštěl bublinky.

Začátek je dost obtížný, to nemá smysl zastírat. Chceme-li řídit činnost mikroprocesoru, musíme samozřejmě vědět alespoň něco málo o tom jak mikroprocesor funguje. Na té nejnutnější minimální úrovni to sice není žádná velká věda, ale přece jen je nutné vstřebat dost nových faktů což samozřejmě vyžaduje jisté intelektuální úsilí. Další obtíž

představuje zdánlivě enormní množství assemblerovských instrukcí. Je jich mnohonásobně víc než příkazů ve vyšších programovacích jazycích

Chce to však jen trpělivost a k ní navrch tvrdohlavou zarputilost. Co člověk nepochopí napoprvé, pochopi možná při druhém, možná při třetím čtení. A pak jednoho dne dojde k poznání, že všechna ta zdánlivá složitost je vlastně velice prostá. Ve chvíli kdy se s ním už sžije, mu assembler připadne možná snadnější než Basic.

Což je ovšem zase omyl vyplývající z prvních úspěchů. Snadno se dá napsat pár řádek trochu věšti program už bude mnohem obtížnější. Ale o tom vám víc než tahle knížka řeknou týdny měsíce a roky praxe.

Možná jste si všimli, že jsem nedodržel běžnou ‚předmluvařskou' praxi a nesliboval nic snadno a rychle. Knížka vás nenaučí, ta jenom pomůže, naučit se musíte sami.

Podle mého názoru je také především na čtenáři, aby posoudil hodnotu knihy kterou drží v ruce. To nezáleží na názoru odborníků ‚přes počítače', kteří měli vůči rukopisu spoustu výhrad protože autor není‚z branže' a nerespektuje zejména terminologickou doktrínu. A nezáleží to ani na názoru lidí od tisku, kteří zase autorovi vyčítali, že není profesionální publicista. Kolem téhle knížky zkrátka bylo dost hádání ještě před tím než se začala tisknout - a možná ještě bude. Leckdo si ji možná koupí jen proto aby mohl kritizovat

Je totiž velice nezvyklá alespoň u nás. V počítačově vyspělejších zemích je literatura psaná neodborníky pro neodborníky už dávno populární, protože specialisté obvykle nebývají schopni vysvětlit všechny složitosti svého oboru lidem bez předběžného vzdělaní. U nás se to však zatím jaksi nenosí. Podle mého názoru je i tato skutečnost důsledkem jistého zaostávaní. Až pro nás budou počítače samozřejmostí, nebudou nám takové knihy připadat nezvyklé. Nakladatelství Mlada fronta skutečně usiluje o to aby jich bylo víc a mimo jiné i proto vznikl program START, v jehož rámci bude počítačová literatura vycházet.

Myslím, že je dobře, že takováto knížka vychází. Nepochybuji o tom že bude potřebná a užitečná.

Jiří Franěk

PŘEDMLUVA AUTORA

Vážený čtenáři,

když jsem stál před úkolem jehož výsledek držíš v ruce, sváděl jsem souboj nejen se stavbou celé koncepce učebního materiálu a volbou formy jeho podání, ale i s u nás vžitým způsobem zpracování obdobné tematiky na stránkách odborné literatury

Nedalo mi to a se svým prvním rukopisem jsem běžel za odborníky. Jejich vyjádření lze shrnout do věty: "Docela hezky, ale chybí tam tohle a ono třeba externí komunikace na příkladu počítačové sítě a technologie výroby nových polovodičových vrstev a multitasking a transputery a taky by tam měl být komentovaný výpis textového editoru nebo databanky a ukázky práce s jinými mikroprocesory a srovnat assembler s dalšími jazyky a vůbec; víš je to takový nevědecký, chybí tomu matematicky formulované podloží a důkazy. A vůbec — proč se ještě zabýváš osmibitovým mikroprocesorem?"

Nedalo mi to a běžel jsem za doktorem a uklizečem a právníkem a topičem a dalšími kteří se chtěli dostat za Basic. Jejich reakce byla odlišná. "Člověče, já už podle ty tvý knížky assemblerem namaluju na obrazovce čárku. A to jsem si myslel bůhvíjak nedostupná věda to je. A syn se díky tobě stal králem kroužku mladých techniků. Představ si, můj kolega mi za ni nabízel pět set. Nemáš ještě nějakou?" Jeden renomovaný redaktor se po přečteni prvotního tvaru dílka hluboce zamyslel, řka: "Podívejte, já to vidím takhle - vyjit by to mělo. Ale ten začátek je příliš hustý. Vy jste to dával číst lidem, kteří po tom prahli. Jenže když to bude na pultu knihkupectví, koupí si to za nějakých XX korun kdokoli. A když těm, kteří na samotné programováni v assembleru nebudou mít buď čas nebo koncentraci, nedá něco aspoň první kapitola, budou těch XX korun najednou cítit jako velkou ztrátu. A přece byste nechtěl, aby v prodeji vaší knihy soupeřil n. p. Kniha s antikvariátem jak v podobných případech někdy býváme svědky.

Dal jsem si to všechno dohromady a přidal zamyšlení o tom, jak se před nějakou dobou stal automobil celospolečenským fenoménem. Kdybych psal knihu o jeho užívání v partii věnované řazení rychlosti by mi technik formule 1 radil popsat převodovku se všemi jejími kolečky a ozubeními, ocelář by požadoval pojednání o odliti skříně, strojař o opracování hřídeli na automatické lince, konstruktér o stavbě spojky, chemik o synteze oleje. Kdo chce a baví ho to, tráví volný čas pod podvozkem svého vozu. Míru své odbornosti si určuje sám a podle toho volí nákup potřebné literatury, ať už je doktorem, zelinářem, úředníkem či novinářem.

Vzpomněl jsem i svých assemblerových začátků — jak nelehko jsem sháněl alespoň útržkovité informace o funkcích jednotlivých instrukcí, kolik assemblerových výpisů jsem musel vyluštit než se mi počalo ozřejmovat jak to vlastně všechno pracuje a **jak** by mi tenkrát nějaká ucelená, souhrnně podaná informace byla tou nejpomocnější podanou rukou. A vím že ji dodnes hledají mnozí z těch, kteří chtějí od svého počítače něco víc, než jim může poskytnout herní automat.

Počítač se stal společenským fenoménem přístupným příslušníkům všech vrstev obyvatelstva reprezentantům všech druhů povolání a zaměstnání. Publikace se obraci ke všem, kteří chtějí poznat základy programování ve strojovém kódu resp. assembleru mikroprocesoru Z80. Proto je zvolena co nejpřátelštější forma podání tématu Učebnice v symbolickém souzněni s názvem programu START není odlehčeným čtením do tramvaje, ale ani nesupluje vysokoškolská skripta. Postup a forma výkladu byly vyladěny tak,

aby čtenář v průběhu výuky nenarážel na černé díry. Prakticky bylo potvrzeno, že každý, kdo si text postupně projde, může začít psát jednoduché programy v assembleru Z80. Pokud se čtenář ještě nesetkal s instrukčními soubory jiných mikroprocesorů, po osvojeni učební látky mu nebude činit větších problémů přechod na kterýkoli z nich.

Učebnice je rozdělena do tří částí.

První je věnovaná začátečníkům, kteří neznají základy toho, jak to v počítači obecně funguje a vypadá, co je to binární logika, programové řízení, apod. Jim především lze jen doporučit, aby si tuto část bedlivě prošli, protože bez znalostí v ní obsažených se nelze pustit do efektivního studia vlastních instrukcí assembleru.

Ve druhé části se čtenář postupně seznamuje s jednotlivými instrukcemi mikroprocesoru Z80. Podrobný výklad každé z nich je doplněn řadou ověřených, bohatě komentovaných příkladů původních i vybraných z odborné literatury obdobného zaměření. Na těchto příkladech si každý přímo na obrazovce počítače může procvičit a ověřit funkce všech instrukcí assembleru Z80.

Všechny programové příklady jsou voleny nezávisle na typu počítače. Jsou tedy aplikovatelné na jakémkoli počítači s mikroprocesorem Z80. Nezbytným předpokladem aplikace je základní programové vybavení pro tvorbu programů v assembleru - editor pro zápis programu spolu s generátorem pro překlad assembleru do strojového kódu a alespoň jednoduchý monitor pro výpis obsahu paměti a ladění programu. Těchto systémových programů je celá řada. Protože se ve svých funkcích i obsluze liší jejich ovládaní, je nutno se naučit podle manuálu vydaného výrobcem. Třetí část přináší praktické rady pro vlastní programování. Tabulky a kresby v příloze obsahuji základní údaje, bez kterých se během tvorby vlastních programů neobejde ani zdatnější programátor.

Omezený rozsah publikace nedovolil uvést vše co s programováním v assembleru souvisí. Tak v ní např. nenajdete informace o podmíněných překladech makroinstrukcí, zakládaní a využívání knihoven programů, ladících a jiných speciálních technikách. Rovněž nezbylo místo pro ukázky programově řízeného přenosu dat mezi počítačem a externími zařízeními atd. Touto širokou problematikou se nepochybně budou zabývat další publikace, které postupně vyjdou v Mladé frontě i jinde.

Na závěr bych chtěl poděkovat lektorům učebnice za podnětné připomínky a nápady. Věřím že díky laskavé péči nakladatelství Mladá fronta se tato publikace stane vítaným pomocníkem všech, kteří chtějí proniknout do tajů programovaní v jazyku, který je počítači nejbližší.

Autor

1. ČÁST

TÝDEN V PŘEDSÁLÍ SOFTWAROVÉHO PALÁCE

BITOVÉ PONDĚLÍ S BYTY A BAJTY

Přišel za mnou přítel který šťastně proplul úskalími Basicu podle manuálu přiloženého k počítači:

„Prosím tě, co je to vlastně ten bit a bajt, jak vypadá adresa? Může to být normálnímu člověku vůbec k něčemu dobré?“

Protože je to přítel a jedním ze zákonů přátelství je pomáhat si navzájem, rozhodl jsem se vysvětlit mu vše jak nejlépe svedu.

„Ale, prosím tě, mluv se mnou lidsky, žádné hieroglyfy. Jde mi o pochopení podstaty, ne o souhrn encyklopedických pouček.“

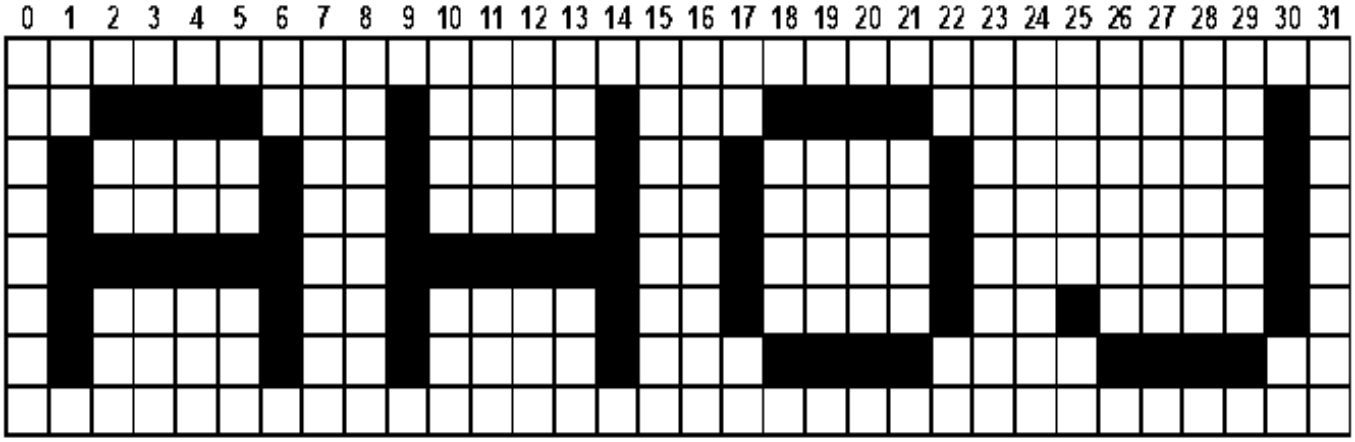
Chvíli jsem uvažoval kudy na to. Stejně jako platí, že prožita zkušenost jednoho je nesdělitelná tomu, kdo ji v alespoň nějaké období rovněž neprožil, musí platit, že snahu o sděleni jakékoli zkušenosti podpoří, bude-li osvěta vycházet z něčeho, co si druhý člověk dovede představit, protože se s tím takřka běžně setkává. A co to je v případě počítače? Přece obrazovka!

„Podívej, tady si namalujeme dům, přesněji jen jeho jednu stěnu. Dům bude mít třeba 192 pater. V každém patře bude 256 čtvercových oken, která se budou vzájemně dotýkat na všech stranách. Prostě taková prosklená stěna obrovského domu.“

Oproti zvyklostem provedu jednu změnu. Patra nebudeme počítat zdola nahoru, ale obráceně. Nejvyšší patro si označíme jako nulté. Číselně vyjádřeno je celkový počet pater našeho domu 192 v intervalu od 0 do 191. Okna pod sebou budeme chápat jako sloupce. Vlevo bude sloupec nultý, vpravo 255. Každé patro má tedy 256 oken v číselném intervalu od 0 do 255.

Je hluboká noc, žádné okno nesvítí. Pohled, který nezaujme. Náhle se v levém horním rohu domu rozsvítí okna tak, že tě pozdraví zřetelným nápisem AHOJ. Zase si to názorně namalujeme. Svítící okna si jasně vyznačíme:

ČÍSLA SLOUPCŮ OKEN



Jak vidíš, nápis AHOJ zabral patra 1 -6 a sloupce 1 -30. Protože však písmena musejí dýchat, tedy musejí být oddělena od svého okolí mezerami, je pro nápis skutečně zapotřebí 8 pater v intervalu 0..7 a 32 sloupců v intervalu 0..31. Každé písmeno se zobrazuje v síti 8x8 oken.

Ty teď sedíš ve stejně velkém, ale setmělém domě naproti tomu, který tě zdraví. Slušnost velí odpovědět na pozdrav. Máš před sebou počítač. Tvou obrazovkou je celá prosklená stěna budovy, ve které sedíš. Aby ses orientoval v tom, co zobrazuje, máš před sebou kontrolní monitor, který ukazuje totéž, *co* stěna oken. Momentálně neukazuje nic, nesviti žádné okno. A tvůj počítač nemá v sobě zabudován ani interpret Basicu, abys věc mohl vyřídit příkazem PRINT „AHOJ", není k mání ani pascalovské WRITE („AHOJ"), nic podobného. Můžeš použít jen strojový kód nebo assembler."

„Brr, strašná bezradnost. Asi bych si namaloval ten předchozí plánek a běžel nahoru rozsvítit v oknech, kde jsou namalované křížky."

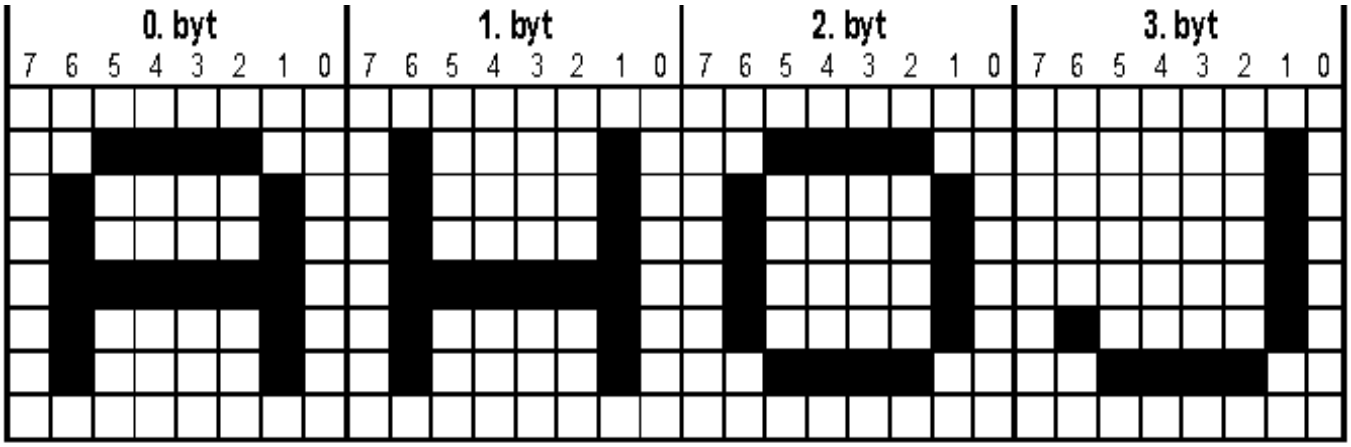
„Pokud by byly v jednotlivých místnostech domu mechanické přepínače, šlo by to. Ale čilejší konverzaci s protějším domem bys takhle *moc* dlouho vést nemohl. A jsme u jádra věci. Využití assembleru je plně efektivní pouze tehdy, když toho o systému počítače, na němž programuješ, víš co nejvíc. Jinými slovy- assemblerem můžeš programově zasahovat přímo do hardwaru ('vnitřností') počítače i jeho externích zařízení tak jako žádným jiným jazykem. Základním předpokladem zvládnutí řečeného je samozřejmě znalost toho, *co* je to bit, bajt, adresa, port, mikroprocesor a jeho registry a takty a přerušeni a..."

„Počkej, počkej, tonu ve tmě domu, z něhož chci odpovědět na pozdrav. Neuvrhuj mě do tmy ještě hlubší. Co tedy dál?"

„Promiň. Vrátíme se k našim nákresům a trochu je přeorganizujeme:

32 bytů po 8 oknech
v každém patře

0. patro/0. byt	**0. patro/31. byt**
192 pater	
191. patro/0. byt	**191. patro/31. byt**



Změnili jsme jen organizaci každého patra. Ta teď obsahují 32 bytů (interval 0 . .31) po osmi oknech (intervaly 0.. 7). Pro nápis AHOJ vlevo nahoře potřebujeme tedy po 4 bytech (bytové sloupce 0..3) v každém z osmi pater (patra 0 .. 7).

Zastavme se na chvilku u oken každého bytu. Každé z těchto osmi oken představuje základní informační jednotku, která se jmenuje bit podle anglického BInary digiT (česky binární, neboli dvojková číslice). Jak už jistě tušíš, psanou podobnost mezi českým slovem byt a anglickým byte (čte se bajt), jsem zvolil záměrně. A jsme u jednoho ze základních vztahů výpočetní techniky - 1 byte obsahuje 8 bitů (a nikdy jinak!). V dalším výkladu budu používat fonetického přepisu slova byte na bajt, jak se vžil už nejen v češtině a slovenštině, ale i v dalších slovanských jazycích. U nás se před zdomácněním bajtu užívalo normované slovo slabika, na které narazíš především v ryze technických textech. Jde o podobný souboj synonym jako mezi videem a magnetoskopem.

Dobře jsi řekl, že bys mohl běhat po místnostech a přepínačem zapínat nebo vypínat světla v oknech. Tím jsi vyjádřil podstatu věci - stejně jako světlo žárovky buď svítí nebo ne (podle toho, zda zapneme nebo vypneme přívod proudu), tak i každý bit má jen dva stavy 0 ('vypnuto') nebo 1 ('zapnuto). V samotných obvodech počítače jsou tyto dva stavy vyjádřeny dvěma různými úrovněmi napětí na bistabilních klopných obvodech, které se na popud zvenčí překlápějí z jednoho stavu do druhého. Do těchto obvodů zapisujeme a z nich čteme jejich logický stav pomocí logických hradel (vstupních a výstupních). V počítačích se používají obvody s různou logikou, nejčastěji logikou TTL (transistor-transistor logic). Proto se stavy 0 a 1 doplňují výrazem logická, zkráceně se pak značí jako log.0 a log.1. V obvodech TTL logické 0 odpovídá napěťová úroveň 0-0,4 V, logické 1 napětí 2-5 V."

„Aha, takže já musím nějak programově dát vědět určitému hradlu, že se má překlopit do stavu logické 1, když chci, aby se určité okno ve stěně domu rozsvítilo, že? Ale jak mu to mám sdělit?"

„Ne vždy se aktivuje funkce hradla úrovní logické 1. Může být aktivována i úrovní logické 0 - tehdy mluvíme o inverzní logice. Značí se vodorovnou čárkou nad názvem funkce, nebo předponou non.

Než dojdeme k adresování, které je jedním z předpokladů úspěšného posílání vzkazů (počítačově řečeno dat), musíme si říci vše potřebné o organizaci bitů v bajtu. Jak sis na posledním nákresu všiml, jsou okna-bity každého bytu-bajtu číslována zleva doprava sestupně od 7 k 0. Tam, kde na nákresu okno svítí, je bit ve stavu logické 1 (na jim reprezentovaném hradle je tedy vyšší úroveň napětí). Kde křížek není (okno nesvítí), je bit ve stavu logické 0 (na jeho klopném obvodu je nízká úroveň napětí).

Abychom se opět o něco přiblížili řeči programátorů, přejmenujeme si naše patra na řádky. Tak např. bajt na řádce 0, v bajtovém sloupci 0 má tuto bitovou reprezentaci:

bit	7	6	5	4	3	2	1	0
stav	0	0	0	0	0	0	0	0

Výsledná číselná reprezentace bajtu, v němž jsou všechny bity ve stavu logické 0, je rovna nule. Pojďme hned o řádku níž (je na ní stříška písmene A):

bit	7	6	5	4	3	2	1	0
stav	0	0	1	1	1	1	0	0

Řeknu ti předem, že číselná reprezentace tohoto bajtu je 60. Jak jsem ji vypočítal? Není to nijak složité, ale chce to cvik. Ze školy jsme zvyklí počítat jen v desítkové soustavě a jakákoli jiná je pro nás nepřirozená, jde nám ‚proti srsti', přestože podstata je stejná. Teď se pekelně soustřeď:

-	-	2^5	2^4	2^3	2^2	-	-
0	0	1	1	1	1	0	0

= 32+16+8+ 4= 60

Vidíš, že bitů ve stavu logické 0 si 'nevšímáme'. U těch, které jsou ve stavu logické 1, umocníme číslo dvě (jde přece o číselnou soustavu se základem 2) číslem jejich pozice v bajtu a výsledek sečteme. Nejvyšším číslem, které může bajt reprezentovat (když jsou všechny jeho bity ve stavu logické 1), je 255. Kolik tedy čísel celkem může bajt vyjádřit ve všech svých bitových kombinacích jedniček a nul?"

„255. Ne, počkej, zapomněl jsem na nulu. Takže 256."

„Velmi správně sis vzpomněl právě na nulu. Ta je totiž při programování snad tím nejdůležitějším číslem vůbec. Hodně věcí se kolem ní točí. Ale o tom až později. Dám ti ještě jeden .louskáček' - kdy bajt vyjadřuje liché číslo?"

„Protože vím, že všechny mocniny čísla dvě jsou sudé, tak snad... Ale ne, vždyť nejnižší pozice je nultá; a nenulové číslo mocněné nulou je vždy rovno jedné. Takže bajt je lichý vždy, když jeho nejnižší bit je ve stavu logické 1."

„Ano. A to platí i pro vícebajtová, tzv. multibajtová čísla. Ale i o nich až později. Dobře jsi použil výrazu nejnižší bit. Říká se mu také nejméně významný, ale to je významově značně zkomolený překlad anglického výrazu *least significant bit.* Když místo *least* (nejméně) uvidíš v počítačové latině, kterou dnes angličtina je, napsáno *most* (nejvíce), pak věz, že se jedná o bit nejvyšší - v bajtu je to bit 7. Protože mikroprocesor může ovlivňovat nejen stavy jednotlivých bitů v bajtu, ale pracovat i s jeho nižší a vyšší polovinou zvlášť, byl pro vyšší 4 bity zvolen anglický termín *gigy nibble* (horní kousek) a pro nižší *low nibble* (dolní kousek). Česky se jim někdy říká vyšší a nižší půlbajt. Abychom se domluvili, uděláme si ještě pořádek ve vyjadřování. Od této chvíle říkejme vždy bit 0 či bit 3, bit 7 apod. Když se např. řekne první bit, může dojít k nedorozuměni, protože se vlastně přesně neví, zda se mysli bit 0 jako 1. bit v bajtu, či bit 1, který je druhý."

„Neměl bych už přece jen odpovědět na ten pozdrav?"

„Teď už víš, co je to bit a bajt. Ale ještě zbývá probrat, jak dát vědět tomu kterému bitu toho kterého bajtu, zda má být ve stavu logické 0 či logické 1. Pokud jsem trochu nešikovně řekl zbývá, pak to znamená naučit se a procvičit obsah všech zbývajících stránek knížky až do konce. Metaforicky řečeno - velmi precizně synchronizovaný ping pong jedniček a nul je vlastně vše, co se v počítači odehrává. Celé programování ve strojovém kódu je takový stolní tenis, při němž držíme pálku (náš program) v ruce a snažíme se nejen o to, abychom nevypadli ze hry, ale aby míček poskakoval tak, jak v tom kterém momentu potřebujeme.

Aby ses mohl obrátit na konkrétní bajt a požadovaně změnit stavy jeho bitů, bajty v paměti počítače musejí být nějak zorganizovány, pojmenovány. Podobně jako když listonoš nese dopis panu Novákovi, kterých má ve svém rajónu dvacet. Pokud by na dopise nebyla uvedena přesná adresa, nastala by svízel. Obsah dopisu (binární data) by se nedostal (data by nebyla přenesena) ke svému adresátovi (do patřičného bytu-bajtu) včas nebo taky vůbec ne. Mohlo by dojít i k doručení jinému adresátovi. S takovou dezorientací a dezorganizací bychom se u počítače - stejně jako v kterékoli jiné činnosti -nikam nedostali. Proto jsou jednotlivé bajty uloženy na jednotlivé, jmenovitě přesně určené adresy. V mikroprocesoru se zase ukládají do jmenovitých, přesně určených jedno - či dvoubajtových registrů.

Na každé adrese je uložen JEDEN bajt se svými OSMI bity. Osmibitový mikroprocesor je schopen adresovat 65536 adres v intervalu od 0 do 65535. Jinak řečeno - takovýto mikroprocesor může adresovat paměť v uvedeném rozsahu adres; je to jeho 'poštovní rajón'. K adresování tohoto rozsahu by nám jeden bajt se svými 256 kombinacemi nestačil. Proto k němu přidáme ještě jeden. U čísla dvoubajtového řadíme vedle sebe 2x8, tedy 16 bitů:

	Vyšší bajt								Nižší bajt							
bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
stav	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Každému ze 16 bitů je přiřazena jeho pozice (pořadové číslo) v intervalu 0..15. V uvedeném příkladu jsou všechny bity ve stavu logické 1, proto u všech číselný základ 2 mocníme číslem jeho pozice. Součet dá výsledek 65535. Jak vidíš, výsledek je lichý, čímž je potvrzeno, že i u multibajtových čísel stav nejnižšího bitu rozhoduje o tom, zda bude celé číslo sudé či liché. Podobně jako samotný bajt má dvě půlky - nižší a vyšší 4 bity - u dvoubajtového čísla rozlišujeme nižší a vyšší bajt."

„Teď jsem z toho trochu jelen. Na jednu stranu mluvíš o tom, jak musí být všechno perfektně proorganizováno, a najednou se tu nějak samovolně množí bajty a bity..."

„O tom, v kterém momentu (programovém kroku) s kolika bajty bude počítač pracovat, rozhoduje (až na některé výjimky) sám programátor. Ten musí svůj záměr přesně uvést ve svém programu. Pak k žádné dezorganizaci dojít nemůže. Pokud k ní vinou programátora dojde, stát se může leccos. Od špatné, tedy chybné interpretace dat s nepředpokládatelnými výsledky, až po programový kolaps. Použil-li jsem výrazu chybná interpretace, pak jde o pohled ze strany programátora, nikoli počítače - ten (je-li funkčně v pořádku) interpretuje, tedy zpracovává všechna data vždy správně. A je mu zcela jedno, zda tebou vložený program zničí sám sebe či zda bude pracovat, jak požaduješ. Filozoficky řečeno - pro něj neexistuje žádné dobro a zlo."

„Takže teď mi bude stačit vědět, jaké číslo má adresa, na níž je bajt, jehož bity chci nějak ovlivnit. Je to tak?"

„Zjednodušeně řečeno je to tak. K větší kompletnosti tématu nám chybí povídání o dvou sběrnicích - adresové a datové. A pochopitelně o mikroprocesoru samotném. Zabývat se tím ještě budeme, teď jen stručně. Aby data, která pošleme po datové sběrnici, přišla na správných osm bitů bajtu, musíme ho adresovat. Když známe jeho adresu, programově vyzveme mikroprocesor, aby nás na ni 'zacílil' adresovou sběrnicí a po datové sběrnici na tuto adresu pošleme data, která provedou s paměťovými buňkami dané adresy potřebné změny. Tak pomocí adresové sběrnice provádíme výběr adresy, jejíž obsah chceme změnit, tedy na tuto adresu něco zapsat. Podobně postupujeme, když chceme přečíst její obsah, tedy to, co už je na ní uloženo. Při zápisu je zapsán momentální stav osmi vodičů datové sběrnice do vybrané adresy, zatímco při čtení se obsah adresy objeví na datové sběrnici, z níž jej můžeme přečíst. Tak po datové sběrnici přenášíme data do adres paměti nebo z nich."

„Už chápu. Když budu znát čísla adres, na kterých jsou bity korespondující s okny, která chci mít v nápisu rozsvícená, postupně na ně pošlu data, která učiní žádané. Ale jak ta data budou vypadat, jak je určím?"

„Analogický postup k tomu, kterým jsme před chvílí zjišťovali, jaké číslo reprezentuje určitý bajt, se přímo nabízí. Místo abys jeho hodnotu dešifroval, sám si ho zkonstruuješ, případně vypočteš. Na to lze jít více způsoby. Nejvíc záleží na tom, čím vším disponuje programovací jazyk, s nímž pracuješ. Ale vraťme se k věci samotné. Představ si obrazovku, kde bajt 0 na řádce 0 má adresu 0. Další bajt vpravo vedle něj má adresu 1 atd. Bajt 0 na řádce 1 pak bude mít adresu o 32 vyšší než bajt nad ním. A tak dál. Kolik adres v jakém číselném intervalu zabere celá obrazovka?"

„32 adres na řádce, řádek je 192, to je"... 6144 adres celkem. A bude to v intervalu 0.. 6143. Teď mě napadá, že při 256 bitech na každé řádce je celá bodová síť takovéto obrazovky 256 x 192, tedy úctyhodných 49152 bodů!"

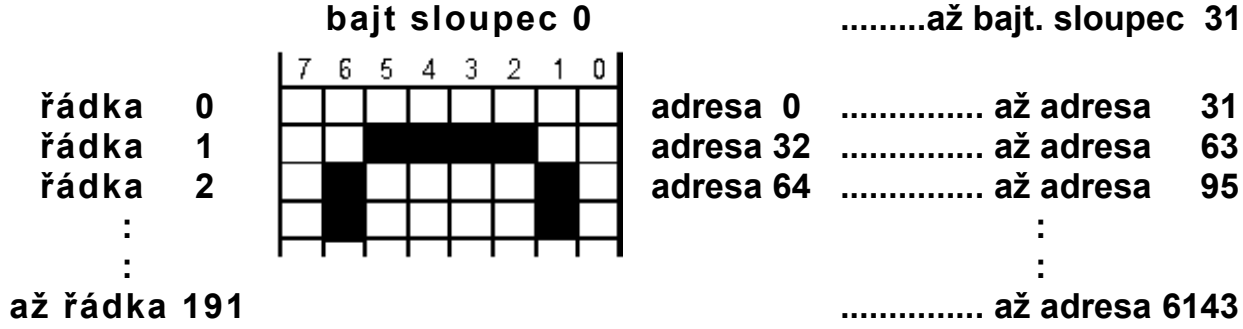
„Správně. Rozlišovací schopnost obrazové paměti, tedy počet obrazových (říká se

i grafických) bodů, je u novějších mikropočítačů ještě o dost vyšší. Průměr se dnes pohybuje kolem 128000 bodů. Teď si představ, že chceš, aby se rozsvítila řada osmi oken na domě zcela vpravo dole. Na jakou adresu paměti pošleš jaká data?"

„To je samozřejmě poslední adresa celé obrazové paměti, tedy 6143. A aby svítilo všech 8 oken, musejí být všechny bity bajtu na adrese 6143 ve stavu logické 1. Číselná reprezentace takového bajtu je 255. To znamená, že na adresu 6143 pošlu bajt s hodnotou 255. Tak se rozsvítí všech 8 oken. Aha, teď už mi dochází, že si napřed musím rozkreslit, co kde na obrazovce budu chtít mít. Zároveň musím vědět, která adresa koresponduje se kterými osmi bity, přičemž každý určitý bit odpovídá každému určitému obrazovému bodu. Bity, které budu chtít aktivovat, budou na úrovni logické 1 a naopak. Z toho všeho si pak vypočtu, kolik který bajt číselně reprezentuje, a výslednou hodnotu každého pošlu na jemu určenou adresu. To je ale oproti Basicu hrozně složité."

„Nesuď dne před večerem. Samozřejmě, že takhle žádný programátor při projekci nápisů na obrazovku postupovat nebude. Pro tyto účely vytvoří rutiny ve strojovém kódu, které tohle všechno budou dělat za něho. Jenže díky tomu, že je nebude zdržovat žádný interpret Basicu, budou rychlé, v rychlosti omezené prakticky jen pracovním kmitočtem mikroprocesoru a schopnostmi programátora. Teď už si alespoň platonicky umíš poradit s okenní korespondencí dům - dům. Poslední kontrolní otázka - co se stane, když na adresu 32 našeho domu s nápisem AHOJ pošleš bajt s hodnotou 0?"

„To je osm oken hned pod okny vlevo nahoře. Schematicky:



Když tam pošlu nulu, přepnu všechna hradla, která byla na úrovni logické 1, na úroveň logické 0. To jsou hradla adresy 32, reprezentující bity 5,4,3,2. Ta, která předtím měla úroveň logické 0 (bity 7, 6,1, 0), na ní zůstanou i nadále. To znamená, že všech osm oken ve sloupci 0, na řádce 1 bude zhasnutých. Čili se vytratí střecha písmene A a zbyde z něj jen jakési zmenšené H."

„Výborně. Na doplnění - adresy obrazové paměti mikropočítačů nikdy nezačínají na adrese 0. To byla jen taková básnická licence pro zjednodušení věci. Proč je tomu jinak, má řadu důvodů. O některých si povíme dál. Stejně tak organizace paměťových buněk ve vztahu k jimi reprezentovaným pozicím na obrazovce se počítač od počítače může různit. Ale teď si už dokážeš představit, že bity obrazové paměti (anglicky *Display File* - soubor dat displeje) jsou jedním z největších polykačů paměti počítače. Čím vyšší rozlišovací schopnost obrázku, tím větší část paměti mikropočítače je vyhrazena pro jeho projekci."

„Teď mě napadá - co je na ostatních adresách paměti, které nevidím na obrazovce, tedy na těch, které jsou mimo adresy obrazové paměti?"

„Tvoje otázka si žádá upřesnění. Především je třeba si uvědomit, že obrazovka je naprosto nevyhnutelný nástroj komunikace mezi uživatelem a počítačem, přesněji - naši částečné kontroly nad tím, co se v počítači odehrává. Ptalo se mne tuhle jedno děvče, k čemu je vlastně obrazovka dobrá, když se všechno odehrává v počítači, nikoli v televizoru či monitoru. Místo odpovědi jsem jí vypnul monitor a řekl, ať na počítači napíše program pro výpočet obsahu trojúhelníku. Vše bylo rázem jasné. Tím chci říct asi tolik, že do každého programu, jehož vývoj či mezivýsledky jeho operací potřebujeme sledovat, musejí být zařazeny i rutiny, které umějí v patřičných momentech převést námi (nebo i operačním systémem či interpretem tvého oblíbeného Basicu) vybraná data na nám srozumitelné znaky (čísla, písmena a jiné symboly), resp. je nějak graficky vyjádřit. Výsledek tohoto převodu, který je smysluplnou, programově určenou skladbou bitů jednotlivých bajtů, je ukládán na patřičné adresy obrazové paměti. Obrázek, který vidíš na obrazovce monitoru či televizoru, je elektronickou reflexí obsahu této paměti. Tak vlastně přizpůsobujeme komunikaci s počítačem našim přirozeným čidlům - v tomto případě očím. Stejně jako počítač, i člověk má svůj systém zpracování informací. I když jsou si obě zpracovatelské struktury ve velice (převelice) hrubých obrysech podobné, přímý přenos informací mezi nimi (alespoň dnes) probíhat nemůže. Kdybychom si mohli strčit jeden konec drátku do nosu, druhý do počítače a sledovat tak, co všechno se v něm odehrává, zobrazení na monitoru bychom nepotřebovali."

„Znamená to tedy, že na ostatních adresách paměti je uložen program, který něco vykonává a jehož požadované průběžné či konečné výsledky jsou s jeho pomoci převáděny na nám srozumitelné vyjádření?"

„Ano, až na to, že program nemusí být nutně na všech ostatních adresách paměti. Většinou spočívá jen v některé části, kterou určí programátor. Ostatní adresy jsou mimo paměťovou mapu vloženého programu, i když hardwarově jsou neustále připraveny k okamžitému programovému využití. Z toho vyplývá, že programy mají různou délku a bývají uloženy na různých adresách paměti. Hradla obrazové paměti se principiálně nijak neliší od hradel její ostatní části - tak vlastně můžeme program uložit i na adresy obrazové paměti. Jejich obsah se nám ovšem bude promítat na obrazovce, na níž (pokud nezvolíme stejnou barvu papíru a inkoustu) uvidíme zdánlivě nesmyslné shluky bodů. V hantýrce se jim říká 'čínské písmo'. Počítač ovšem za nic nemůže - jen svědomitě promítá to, co má na dotyčných obrazových adresách uloženo. Takovýto případ využití obrazové paměti je výjimečný a používá se většinou jen tehdy, když je takové uložení programu či zpracovávaných dat nutné z nedostatku volné paměti pro některé programové mezioperace."

„Takže když píši program v Basicu, určité, pro mne zatím zcela neznámé rutiny zařizují, aby se výpis mých programových řádek zobrazoval na monitoru. Bez těchto rutin bych se ve svém programu neorientoval. Jak ale takové zobrazeni probíhá?"

„Podobně jako ty mne, musím i já tebe občas zadržet v rozjezdu. Přenos obrazových dat z počítače na monitor patří do ‚vyšší dívčí' komunikace počítače s jeho okolím. Ta se dá v programátorské a konstruktérské kuchyni upravit na tisícero způsobů. Projevuje se to i u jednotlivých typů počítačů - každý má trochu jiný operační systém, jiné řešeni vnitřní i vnější komunikace, i když existují jisté dílčí snahy o standardizaci. Přestože

základní zákonitosti platí pro všechny stejné, každý počítač nese 'rukopis' svých tvůrců. Podobně dva lidé nikdy nenapíší naprosto identické programy, i když ve výsledku budou oba dělat (skoro) totéž. A za vším je jen ta logická 1 a logická 0. Vzdáleně asi tak, jako za vší literaturou je jen tužka a papír, ale obsah knih je různý.

Předtím než odhalíme některé z dalších tajů základů výpočetní techniky, bez jejichž znalosti se nelze pustit do programování ve strojovém kódu či assembleru, chci ti připomenout jednu odedávna platnou pravdu. Stejně jako spisovatelem není každý, kdo zná pravopis, fyzikem, komu spadlo jablko na hlavu, matematikem, kdo si umí přepočítat výplatu, tak programátorem se nestane nikdo jen tím, že bude schopen sypat jazykové příkazy z rukávu.

Vlastní zápis programových instrukcí je v procesu tvorby programu na posledním místě. Nemůžeme přece obdivovat jen samotná tři písmenka, jednu číslici a rovnítko v Einsteinově formuli $E = mc^2$. Obdivuhodný je duševní tvůrčí proces, jakým k ní dospěl, a co je nejdůležitější - bez něhož by se k tomuto skvělému výsledku zřejmě vůbec nedopracoval. Pokud tedy má programování nést znaky tvůrčího procesu vedoucího k řešení nějakého problému, je třeba se napřed zamyslet nad problémem vlastním, 'ohledat' jej ze všech stran, dokud se nezačne zdát, že ‚užuž' jsme kápli na to, jakou cestou se dát. Tuto cestu pak musíme postupně algoritmizovat, tedy převádět na (sestupné) fáze dílčích etap řešení. Tak vytváříme jeho strukturu, členitou architektonickou stavbu, kterou nakonec převedeme, rozepíšeme do procedur či rutin toho kterého programovacího jazyka. Aby celý tento tvůrčí proces přinášel očekávané výsledky, je třeba hodně znát. Není náhodou, že drtivá většina tvůrců programovacích jazyků jsou matematici.

Nejhorší chyba, které se 'programátor' může dopustit, je, když po rozhodnutí vyřešit nějaký úkol ihned sedne k počítači a 'od podlahy' začne vyťukávat programové instrukce. Ve své činnosti se začne brzy a spolehlivě utápět. Po dosažení určité hranice kvantity takto vyprodukovaných příkazů už ho čeká jen donkichotský zápas s narůstající změtí nekontrolovatelných vazeb všeho se vším. Nakonec zjistí, že jen ztratil spoustu času, a všechno vzdá.

Můžeš namítnout, že Einstein je jen jeden. Ale fyziků je hodně. Nakonec, i kdybys programoval jen na úrovni rekreačního nohejbalu - tak proč ne? Všichni si rádi poslechneme profesionální hudebníky, ale stejně rádi si zamuzicírujeme amatérsky. Chtěl jsem tě jen upozornit na úskalí toho, do čeho jdeš. I proto, že o programování panují v obecném povědomí trochu mylné představy."

„Začínám mít pocit přetížené informační sítě, hradla paměti se mi překlápějí na logickou 0. Ale už teď vím, že zítra budu mít řadu dalších otázek."

„Jak to tak vidím, dáme si týden průpravy před vstupem do vlastního studia programování mikroprocesoru Z80. Zítra ahoj a nezapomeň vzít čaj - ale normální, ne nějaký digitální!"

ÚTERÝ ZASLÍBENÉ PAMĚTI

Celý den jsem přemýšlel o tom, jak svého přítele navigovat dál, abych mu ve všem neudělal zmatek a aby bolest poznávání byla co možná nejsladší. Hned u dveří mi, trochu nevyspalý, hlásil:

„Člověče, já se ti teď na svůj počítač po včerejšku dívám úplně jinak. A víš, co se mě zmocňuje? Touha tomu všemu přijít na kloub. Abych to všechno mohl vědomě řídit! S tím Basicem ZX Spectra si připadám jak v polovičním bezvědomí."

"Prober se, lomcuje tebou touha vládnout! Ale vždy je lepší být pánem počítače, než jeho sluhou, kterému na prstech tuhnou mozoly z nekonečného honění bizarních příšerek ještě bizarnějšího světa bludišť počítačových her."

„O těch už mi nemluv. Včera jsi naznačil něco o sběrnicích - adresové a datové. Jak to s nimi vlastně je?"

„Napřed si povíme něco víc o paměti. Jádrem vlastní buňky paměti, kterou reprezentuje jeden bit svým logickým stavem 1 nebo 0, je bistabilní klopný obvod. Jak víš, může být vždy jen v jednom ze dvou stavů. O tom, v jakém z nich bude, rozhoduje napěťová úroveň na vstupu paměťové buňky. Jím do buňky vstupujeme s informací (nastavením vyššího či nižšího vstupního napětí), která se v buňce projeví jedním z obou možných stavů - buď se překlopí, nebo zůstane v předchozím stavu. Tak do buňky zapisujeme - nutíme ji, aby podržela určitou informaci. V počítačové technice je to základní informační jednotka - 1 bit.

Podobně jako by nám k ničemu nebylo, kdybychom nemohli číst napsané, je nutné, abychom mohli logický stav, v němž se nalézá paměťová buňka, přečíst. S takto získanou informací pak pracujeme programově dál.

Klasická buňka má čtyři linky. Vstupní pro zápis, výstupní pro čtení a zbývající dvě pro aktivaci jedné z obou funkcí - linku WRITE (zapiš) pro převedení logického stavu z datové sběrnice do vstupu adresované buňky a READ (přečti) pro převedení logického stavu z výstupu adresované buňky na datovou sběrnici.

Adresová sběrnice slouží především rychlému nalezení, resp. výběru momentálně adresované buňky. Když má k takovému výběru dojít, do adresového registru se uloží binární kód hledané adresy. Dekodér adresy jej převede na kód, jímž konstruktér označil jednotlivá místa celé přístupné paměti. Celek každých osmi paměťových buněk (každé místo neboli lokace paměti) je umístěn v matici, jíž jako souřadnice procházejí adresové vodiče ve sloupcích a řádcích. Tyto vodiče umožňují výběr jednoho určitého paměťového místa. Šestnáctibitová adresa je rozdělena na dvě části - vyšší a nižší bajt - jeden pro sloupce, druhý pro řádky. Podle toho, v jakém stavu jsou bity každého z těch-

to bajtů, je pomocí dalších (výběrových) obvodů velmi rychle určeno paměťové místo s danou adresou."

„Takže když je na jedné paměťové adrese osm paměťových buněk — a protože bajt má osm bitů - pak v mém počítači s pamětí 64K je takových buněk 65536 x 8, to je 524288! Přes půl miliónu!"

„A to není na dnešní poměry nijak zvlášť velká paměť. Moderní počítače střední třídy mají paměťových buněk nejméně desetkrát víc. Když si představíš, že ještě před nějakými třiceti lety byla paměťová buňka reprezentována minimálně jednou elektronkou nebo několika relé a změtí drátů a pasivních prvků (rezistorů, kondenzátorů a cívek), pak tvé ZX Spectrum by tehdy zabralo několikaposchoďový dům. Nehledě už na rychlost zpracování dat, ale i na to, že takový počítač by sis nemohl koupit, ani kdybys šetřil celý život. Revoluční skok byl umožněn zavedením objevných technologií výroby polovodičových součástek a jejich následnou integrací (prostorovým zhuštěním) na malé ploše křemíkového čipu s opravdu miniaturní architekturou jeho stavby. Ale odpoutejme se od hardwarových detailů a podívejme se na různé druhy pamětí. Existují dva základní - RAM a ROM.

RAM *(Random Access Memory)* je paměť, s níž můžeme (samozřejmě je-li patřičně napájena z proudového zdroje) vykonávat obě základní operace - zapisovat do ní i číst z ní. Jinak řečeno - do ramky můžeme ukládat data a také je z ní odebírat. Takovouto pamětí je celá volná, tzv. operační paměť, do níž ukládáme program buď z klávesnice, nebo z nějakého paměťového média (pásku, disku apod.) nebo jiného externího zařízení. Ramka má tu nepříjemnou vlastnost, že po odpojení zdroje napájení zcela 'ztratí paměť" - zmizí z ní všechny informace, které obsahovala.

Oproti tomu z paměti ROM (*Read Only Memory)* můžeme pouze číst její obsah, který byl do ní napevno uložen výrobcem. To znamená, že do romky nemůžeš nic zapisovat, můžeš z ní jen číst. I když se můžeš v tomto směru cítit ošizen, není to tak. Do těchto pamětí se ukládají obslužné programy operačního systému počítače (nebo externího zařízení) a stávají se tak jeho přímou součástí. U mikropočítačů obvykle obsahují ještě rutiny pro interpretaci Basicu, který je tak v nich zabydlen natrvalo. Jakkoli měnit skladbu těchto rutin by si mohl dovolit jen opravdový znalec. Těch je mezi běžnými uživateli málo, proto jsou tyto rutiny uloženy do nepřepsatelných romek, a tak zabezpečeny před poškozením. Na rozdíl od ramky, romka po odloučení od napájecího zdroje paměť neztrácí. Vše v ní zůstává zachováno, připraveno k dalšímu použití po opětném připojení napájecího napětí. To je určitá výhoda i z hlediska okamžité připravenosti celého systému. Nemusíš hledat kazetu či disketu se záznamem operačního programu, který bys nutně musel po každém zapnutí počítače ukládat do jeho paměti.

Zvláštním typem paměti ROM je např. EPROM. Je to romka, jejíž obsah je mazatelný ultrafialovým zářením. Když ji zasadíš do počítače, chová se stejně jako romka. Epromku však můžeš na speciálním programovacím zařízení několikrát přeprogramovat podle svých požadavků.

Poslední dobou se objevují speciální ramky, kterým stačí jen velmi malé napájení z miniaturní baterie k tomu, aby podržely informaci do nich uloženou. Používají se jako *tzv.* RAM-disky. Tímto názvem se vyjadřuje, že jde o vnější paměťové médium, z něhož - podobně jako z disku - můžeme do operační paměti uložit požadovaná data. Výhodou RAM-disku je obrovská rychlost přepisu jeho obsahu do operační paměti počítače a

snadný přístup ke kterékoli jeho paměťové buňce, jejíž obsah můžeme kdykoli změnit. Oproti mechanickým paměťovým jednotkám má teoreticky nekonečnou životnost. Bohužel, poměr jeho ceny na 1 bit je stále ještě dost vysoký.

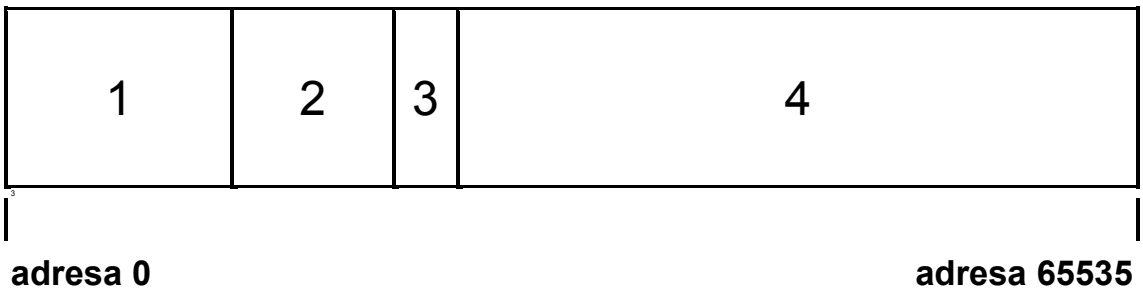
Z hlediska konstrukce se paměti dále dělí na dynamické a statické. Dynamické se opírají o polovodičovou technologii, která umožňuje jejich funkci při malé spotřebě proudu. Jsou to v podstatě paměti kapacitní. Na kondenzátorku uvnitř paměťové buňky je udržováno napětí odpovídající v ní zapsané logické úrovni. Nevítanou vlastností těchto buněk je, že napětí z jejich kondenzátorků se velmi rychle vytrácí. To si vyžádalo zavedení pomocné operace, která ve velmi krátkých časových intervalech tyto unavené kondenzátorky osvěžuje, pomáhá jim udržet jejich potenciál. Podobně jako když artista dodává talířům točícím se na prutech ubývající energii, aby nespadly na zem. Na druhou stranu je výhodné, že pro konstrukci jedné paměťové buňky stačí 3 tranzistory typu MOS s nezbytným kondenzátorkem a místo 4 linek pro čtení a zápis jen 2. Dokonce jsou takové buňky, které mají jediný tranzistor (plus kondenzátor) a společný bod pro zápis i čtení informace!

Statické paměti, které se u mikropočítačů moc nepoužívají, mají výhodu v tom, že netrpí průběžnou sklerózou, typickou pro výše uvedené kolegyně.

Obecně je různých druhů a typů pamětí dlouhá řada, která stále narůstá. Každá má něco svého - přednosti jedné jsou nedostatky ostatních. Volba typu paměti je přímo závislá na aplikačních podmínkách."

„Jaké paměti mám tedy ve svém malém mikropočítači a jak s nimi mohu nakládat?"

„U různých počítačů to bývá dost různé. Svým rozložením ne moc výhodná mapa celé paměti prvních komerčně vyráběných osmibitových mikropočítačů vypadá přibližně takto:



1 - rutiny operačního systému a interpret Basicu (ROM 16K)
2 - obrazová paměť (dynamická RAM - cca 6,8K)
3 - systémové proměnné (dynamická RAM - cca 0,2K)
4 - volná paměť (dynamická RAM - cca 41K)

Paměť RAM má tedy rozsah celkem 48K. Běžnému uživateli vyhrazené hřiště je pod číslem 4. Zná-li víc, může si hrát i na čísle 2. Znalci se postupně zpřístupňuje malé, leč velmi zajímavé hřišťátko 3 -jsou v něm uloženy některé průběžné informace o práci operačního systému počítače. Jejich změnami (leží přece v paměti RAM) můžeme docílit velmi zajímavých i účelných efektů. Hřiště 1 je tabu (paměť ROM). Ovšem superznalec si dokáže poradit i s ním."

„A kam se ukládá můj Basicový program?"

„Jeho uložení je řízeno interpretem Basicu ve spolupráci s operačním systémem počítače. Ve výše uvedené mapě paměti se bude ukládat od prvních adres volné paměti. Možná tě trochu překvapí, že i když programuješ v Basicu, do paměti se skutečně ukládá binárně zakódovaný text. Ten má tvůrcem interpretu předem stanovená pravidla kódování. Podle nich je pak interpret schopen uložený kód dešifrovat a správně jej interpretovat. Uložený kód však netvoři instrukce tohoto kódu, ale je to jen zakódovaný soubor dat. Proto musí být interpretován. Vedle interpretů existují ještě kompilátory jazyků. Zatímco interpret stále dokola interpretuje každý programový příkaz, jako by jej 'viděl poprvé v životě', kompilátor převede (zkompiluje) celý, v tom kterém jazyku vytvořený program naráz do ‚na míru šitých' instrukcí a dat programu ve strojovém kódu - těm už počítač rozumí přímo. Interprety jsou o dost pomalejší než zkompilovaný program. Podobně ani dva lidé, kteří neumějí jazyk toho druhého, se nedomluví bez tlumočníka, jehož překladem se celá komunikace výrazně zpomaluje.

Poprvé jsem se zmínil o existenci instrukcí strojového kódu a datech. Jaký je mezi nimi rozdíl? Podíváme-li se na ně přímo do paměti, pak vůbec žádný. Všechno to jsou pouhé bajty, spočívající na adresách paměti. Uložme třeba na adresu 0 našeho včerejšího domu-obrazovky bajt s číselnou hodnotou 201. V bytu nahoře vlevo se okna rozsvítí takto:

Je tenhle bajt instrukcí nebo jsou to data? To záleží jen na přístupu, jaký k takovému bajtu zvolím. A ten přístup volím programovým příkazem, kterým velím mikroprocesoru, co s tímto bajtem má dělat. Dejme tomu, že mu patřičnou instrukcí přikážu, aby tento bajt přenesl (přesněji - okopíroval) na adresu 1. Po vykonání příkazu budou na adresách 0 i 1 stejné bajty. V obou bytech vedle sebe se bude svítit stejně. Teď ale navedu mikroprocesor na adresu 0 tak, že do jeho programového čítače umístím číslo 0. Mikroprocesor v ten moment obrátí svou pozornost na adresu 0 tak, že bude na ní uložený bajt dekódovat jako jednu z instrukcí svého instrukčního souboru, a bude-li ji znát, provede ji. V tomto případě - převedeno do srozumitelnějšího assembleru - bajt s hodnotou 201 reprezentuje instrukci RET (návrat), což je obdoba basicového příkazu RETURN. (Jak tento návrat provede, spadá do jiné části výkladu.) Jinými slovy - obsah adresy, kterou má mikroprocesor momentálně ve svém veledůležitém programovém čítači, bere vždy jako kód instrukce. Pokud mu taková instrukce přikazuje, aby provedl nějakou operaci s bajtem (bajty) na nějaké adrese (adresách), pak tento bajt (bajty) bere jen jako data. Mikroprocesor Z80 má kolem 700 instrukcí s různou délkou - od 1 do 4 bajtů."

„To znamená, že jakýkoli program v jakémkoli jazyku je sice do paměti ukládán ve formě nějak zašifrovaných dat, ale pokud tato data nejsou přeložena, 'přetavena' do instrukcí strojového kódu, kterým mikroprocesor rozumí, počítač se s takovým programem přímo nedomluví?"

„Ten se domluví vždy, i když taková domluva povede třeba k záhubě vlastního programu. Výsledek takovéto ‚interpretace' programu však pochopitelně půjde totálně mimo náš původní záměr. Takže z našeho hlediska si počítač sice bude hezky povídat sám se sebou, ale my budeme zcela mimo hru. A jsme to právě my, kdo musíme programově vést počítač cestou, kterou sami vědomě připravujeme. Jestliže jsme vymysleli interpretační jazyk, který generuje kód programu, jenž je ukládán do paměti jako data, pak pro jejich dekódováni (chtěnou interpretaci) musíme sami vytvořit správně pracující interpret, který v paměti uložený kód vyluští a provede přesně a jednoznačně to, co mu 'šifry' kódu provést velí. Co tvoří interpret, už asi tušíš sám. Jsou to sledy instrukcí strojového kódu - tzv. rutiny - které vše interpretují tak, jak jim programátor při jejich tvorbě určil.

Kromě assembleru, který je pouhým mnemonickým přepisem jednotlivých instrukcí strojového kódu, se ostatní jazyky nazývají vyšší programovací jazyky. Na samém počátku historie elektronických počítačů se programovalo jen ve strojovém kódu. Programátoři museli přímo na jednotlivé adresy ukládat čísla, která byla programovými instrukcemi a daty tohoto kódu. Už brzy si budeš umět představit, jak obtížně se jim muselo pracovat, o orientaci v takovém programu nemluvě.

Říká se, že kolo vynalezla lenost člověka. První vyšší programovací jazyky měly zbavit programátory otravné manipulace se strojovým kódem a zlepšit orientaci v programu. Historie těchto jazyků je poměrně dost krátká, všechno to jsou ještě mladíci a mladice. Už mají dokonce i své generace časově řazené do období, v nichž byly svými duchovními otci 'vygenerovány'. Podobně mají své generace i operační systémy počítačů a nakonec i počítače samy. V současné době vedou tvůrci programovacích jazyků zápas o zrod jazyků 4. generace. Ty už budou mít aspoň náznak toho, co se skrývá pod pojmem umělá inteligence. Ale již dnes je celkem zřejmé, že na bázi dnešní architektury počítačů bude takový typ inteligence těžko realizovatelný. Jsme tedy ve fázi, kdy požadavky kladené na jazyky jsou zdrojem impulsů pro hledání nových cest konstrukce počítačů a jejich operačních systémů. Velká naděje se vkládá do optických počítačů, s jejichž konstrukcí vědci zatím urputně zápasí. Tato nová technologie umožní, aby v počítači probíhalo množství dějů skutečně současně (nikoli 'přerušovaně-spojitě' jako je tomu dnes) při použití většího počtu logických úrovní. Jde tedy i o hledání zcela nového typu základní logiky, která má u počítače polovodičového jen ty dva nebohé stavy - logickou 0 a logickou 1."

„Jak tě tak poslouchám, uvědomuji si, že jsme vlastně počítačovými neandertálci. Za pár desítek let budou naše Spectra a Amstrady k dostání za vysoký poplatek jen ve starožitnostech a nikdo už na nich nebude umět nic naprogramovat."

„Přejme si, aby to tak skutečně dopadlo. Jinak - kde by tu byl jaký pokrok? Ale zpět do našich dnů. Zítra máme středu a to je nejvyšší čas na probrání některých drobných nepříjemností strojového kódu. Mám na mysli hlavně různé číselné soustavy o různých číselných základech. Tak žádné ponocování u počítače! Svěžest je předpokladem úspěšného zítřku."

HEXADEKADICKÁ STŘEDA

„Hexade ... jakže jsi to nazval dnešní den?" „Bohužel, nedá se nic dělat. A to v tom názvu ještě není obsaženo vše, čím tě dnes budu strašit. Já to radši hned napíšu na papír:

Binární čísla	Dekadická čísla	Hexadekadická čísla
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F
10000	16	10
.	.	.
.	.	.
01111111	127	7F
10000000	128	80
.	.	.
.	.	.
11111111	255	FF

Binární čísla už trochu znáš. Novinkou jsou hexadekadická (v dalším textu budu jejich dlouhý název zkracovat na HD). Jejich číselná soustava má základ 16. Jak vidíš, pomáhají si písmeny abecedy. Je samotná značíme písmenem H (např. 3BFAH), aby-

chom si je nepletli s jinými. Rozluštit, kolik které vlastně dekadicky je, není nijak těžké. Obecně platí to, co jsme si zjednodušeně ukázali už u čísel binárních. Číslo, které je základem dané soustavy, postupně mocníme pořadovou pozicí každé číslice převáděného čísla a obdržený mezivýsledek danou číslicí ještě vynásobíme. Takto získané výsledky nakonec sečteme a dostaneme hledanou hodnotu. Aby to bylo jasné, ozkoušíme si to na dekadickém čísle, třeba **2034,5:**

$2.10^3 x 0.10^2 x 3.10^1 + 4.10^0 + 5.10^{-1} = 2000 + 0 + 30 + 4 + 0, 5 = 2034, 5$

V příkladu použité exponenty **3, 2, 1, 0, -1** jsou pořadovými pozicemi jednotlivých číslic převáděného čísla. To byl ovšem převod do téže soustavy. Všimni si, že nul si v převáděném čísle nemusíš všímat. Zkus sám převést **A9FH** na desítkovou soustavu."

„Strašil jsi mne zbytečně. To je jednoduché:

$10 .16^2 + 9 .16^1 + 15 .16^0 = 2560 + 144 + 15 = 2719$."

„Dobře. Ale teď proveď kontrolu správnosti, tedy převod zpět do HD soustavy." „Hmm, hmm ... nedocenil jsem situaci..."

„I to je jednoduché. Dělí se číslem základu dané soustavy, přičemž důležité jsou zůstatky dělení:

```
pozice 0          2719 : 16 = 169 + .15 (15 = FH)
1            169 : 16 =   10 +   9 ( 9 = 9H)
2                              | ...10 (10 = AH)
```

„Výsledek je **A9FH.** Kontrola správnosti vyšla na jedničku ve všech reálných číselných soustavách. Vše se ti ozřejmí na běžném dekadickém dělení:

```
pozice 0                    365 :10 = 36 + 5        (5)
1                      36:10 =   3 + 6       (6)
2                                  I... 3        (3)
```

Ale pozor na možný omyl z nepozornosti v obdobách tohoto případu:

```
pozice 0                   107 : 10 = 10 + 7       (7)
1                     10 : 10 =    1 + 0       (0) !!!
2                                  I...1       (1)
```

Zkus převést dekadických **255** na dvojkovou soustavu." „Teď už to půjde:

pozice 0	255 : 2 = 127 + 1
1	127 : 2 = 63 + 1
2	63 : 2 = 31 + 1
3	31 : 2 = 15 + 1
4	15 : 2 = 7 + 1
5	7 : 2 = 3 + 1
6	3 : 2 = 1 + 1
7	I … 1

To je našich známých osm svítících oken, tedy bajt 11111111. Ale moc příjemné tyhle převody nejsou. Kde se ve výpočetní technice vůbec vzaly?"

„Mohu tě uklidnit vyzvoněním jednoho sladkého tajemství. Podobně jako nám všem dělá potíže přepočítávání minut na hodiny či vteřiny, jen málokterý programátor počítá v jiných soustavách stejně hbitě jako v desítkové, spíše je proklíná. Ale nemyslí to zase tak úplně vážně. Až budeš programovat ve ‚strojáku' nebo snad dokonce tvořit hardwarové konstrukce, přijdeš na to, že věc má některé přednosti. Počítání ve dvojkové soustavě je v přímé souvislosti s binární logikou výpočetní techniky. Potřeba počítat v šestnáctkové či osmičkové soustavě se odvinula např. z konstrukcí paměťových bloků do matic a jejich adresování. Jedna osmičková číslice v sobě zahrnuje tři, šestnáctková čtyři dvojkové číslice. Převod mezi binárním a šestnáctkovým číslem není pro počítač žádným problémem, není třeba používat komplikované přepočty.

Převážně v americké a anglické literatuře často narazíš na písmeno K za číslem. Třeba 16K znamená 16 x 1024 bajty, tj. 16,384 kB (B je symbolem bajtu, malé b bitu). 1K je tedy 1,024 kB. Malé k je obvyklým označením pro 1000 jednotek. To si lehce potvrdíš nahlédnutím do katalogu s nabídkou polovodičových pamětí. Mikroprocesor Z80 tedy může adresovat paměť v rozsahu 64K, což je 65,536 kB, neboli 10000H bajtů v adresovém rozsahu 0 - FFFFH. U nás se používá zřetelnější forma zápisu - např. 16 KB místo 16K, 4 MB místo 4M a podobně.

Pohledem na sloupce čísel z úvodu dnešního dne snadno zjistíš, že každé čtveřici binárních číslic odpovídá jedna HD číslice. Proto převod mezi binárními a HD čísly není tak obtížný a není nutné jej vypočítávat. Třeba 1000 1000 1000 0001 binárně bude jednoduše 8881H.

Pro programování v assembleru se používají editory assembleru v kombinaci s generátory strojového kódu (říká se jim rovněž assemblery), které umožňují v jednom a tomtéž programu libovolně kombinovat čísla kterékoli z uvedených soustav. Generátor, který vygeneruje a do paměti uloží překlad assembleru do strojového kódu, si s tím, jakož i s mnohým dalším, dokáže hravě poradit. Při programování budeš nejčastěji pracovat s HD čísly FH, FFH, FFFFH, 80H, 4000H, 8000H a čísly o 1 menší nebo větší. Pokud jde o binární čísla, dost pomáhají při práci s logickými funkcemi a při tvorbě znaků (vzpomeň si na skladbu bitů v nápisu AHOJ a představ si, že bys ji měl přepočítávat na jinou soustavu!). Velmi výhodné bývá použití obou těchto soustav při práci s obrazovou pamětí.

Měli bychom si osvětlit jemný zmatek, který se uhnízdil ve vžité terminologii. Slovem assembler se označuje jak jazyk, tak i překladač (generátor), který převádí assemblerový program do strojového kódu. Původně byl tento jazyk pojmenován *assembly lan-*

guage, přeneseně jazyk pro montáž (strojového kódu). Slovo assembler (montér) se užívalo jen pro pojmenování generátoru. Postupem doby se tak začalo říkat i jazyku samotnému. Normovači technické češtiny jej nazvali jazykem symbolických adres. Tento název, který si svou délkou nezadá se jmény polynéských králů, obsahuje vedle slova jazyk jen jednu z jeho předností. Já tento název nepovažuji za příliš trefný, proto budu stejně jako všichni programátoři nadále používat montérský výraz assembler.

Nezbytným doplňkem programování v assembleru je tzv. ladicí prostředek. Podle nejstarších programů tohoto typu se mu v počítačové hantýrce říká monitor. Jím si můžeš prohlížet obsahy jednotlivých adres, měnit je atd. Monitory mají řadu dalších skvělých dispozic. Pomocí těch modernějších můžeš provádět velmi detailní rozbor programu ve strojovém kódu a opravovat jeho chyby - tzv. jej ladit. Zvláště při práci s monitory se ti velmi vyplatí orientace v HD číslech."

„To je, pokud jde o čísla, všechno?"

„Kdepak. Čísla versus počítač je téma nekonečné. Např. jen práce počítače s pohyblivou řádovou čárkou nedává spát celým vědeckým týmům. Další věčná témata - jak operovat s multibajtovými čísly, jak dosáhnout vysokého počtu platných desetinných míst, jak programovat matematické funkce atd., atd. A kdybychom šli až někam do budoucnosti, nepochybně se v ní budou matematici zabývat computerovými hrátkami s nekonečnými čísly. Víš, jaká to pro ně bude slast, až si budou moci doma sednout k pravnukovi Spectra a sečítat nekonečna?

Tebe ještě čeká přeskočení dvou základních překážek - doplňkových čísel (najdeš je až v 'samoučce ', kterou ti předám na závěr našeho seminárního týdne) a logických funkcí - ty jsou jedním z nejdůležitějších fenoménů celé výpočetní techniky. Na ně se podíváme hned zítra, co říkáš?"

„To se pořád ještě nezačne programovat?"

„Hleďme - netrpělivý žák autoškoly, který ucpal křižovatku. Až budeš stát na prahu svého prvního většího programu, hořce vzpomeneš této otázky. Vezmi radši všechnu logiku do hrsti, budeme ji zítra potřebovat oba!"

LOGICKÝ ČTVRTEK PANA BOOLEA

„On ještě někdo přijde?"

„Bohužel, geniální pan Boole se s námi rozloučil už v 19. století. Ale zanechal nám něco, bez čeho by nepracoval ani jeden digitální počítač. Tok bitů v hardwaru podléhá předem stanoveným zákonitostem, jimž kraluje Booleova algebra. Její autor ji objevil v době, kdy vize o počítacích strojích vzrušovala jen několik málo nepokojných a vynalézavých duchů. Všichni ostatní považovali 'hraní si' anglického matematika G. Boolea za výstřelek bez jakéhokoli užitku - nikoli první, ale ani poslední ironie evoluce lidského poznání.

I když Booleova algebra byla matematiky postupně rozpracována do značné šířky, my si povšimneme hlavních tří logických funkcí, které jsou základem operací s bity a bajty (AND, OR, NOT) a doplníme je čtvrtou (XOR) z instrukčního souboru Z80. U každé z logických funkcí je uvedena tzv. pravdivostní tabulka, která udává, jaká hodnota bude na výstupu Q. logického členu při uvedené kombinaci hodnot na obou jeho vstupech A a B (u funkce NOT jen jednom - A):

AND / logický součin/

vstupy		výstup
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR /logický součet/

vstupy		výstup
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

NOT / negace, inverze/

vstup	výstup
A	Q
0	1
1	0

XOR /nonekvivalence/

vstupy		výstup
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

U funkcí AND, OR a XOR si povšimni těchto vztahů:

1. Je-li A=0 i B = 0, pak u všech bude i Q=0.
2. Je-li A=1 i B = 1, pak u AND i OR bude Q=1. jen u XOR bude Q=0.
3. Při rozdílných úrovních A a B u OR i XOR bude Q=1, jen u AND bude Q=0.

Funkce NOT jednoduše převrací hodnotu vstupu.

Tyto vztahy si musíš vrýt do paměti jako malou násobilku. Bude-li se paměť vzpírat, pořiď si kartičku, na níž bude vše poznamenáno. Proč? Protože bez toho se programovat v assembleru nedá. Uvedené funkce jsou v instrukčním souboru mikroprocesoru Z80 přímo implementovány. Můžeme jimi bezprostředně ovlivňovat stavy jednotlivých obvodů paměti, mikroprocesoru a vstupních i výstupních portů. Proto se říká, že assembler je jazyk, který má přímý přístup k jednotlivým bitům a může bezprostředně ovlivňovat jejich stavy. Především v konstrukcích výpočetní techniky mají své místo další logické funkce, které jsou kombinací výše uvedených, např. NOR (NOT - OR), NAND (NOT - AND) apod. Bez znalosti logických vztahů nelze postavit žádný počítač."

„Takže vyšší programovací jazyky můžu považovat za určitý 'diplomatický jazyk', jehož mluvou se dostávám k věci oklikami, zatímco assembler jde přímo na věc?"

„I tak by se to dalo říci - ovšem s přihlédnutím k tomu, že i assembler vyžaduje překlad do strojového kódu. Ale jako v životě obecně, i tady platí, že někdy je užití diplomacie výhodnější a důvtipnější. Proto assembler nelze považovat za něco samospasitelného. Ke všemu, tedy i k programování, je nutno přistupovat s nadhledem a rozvahou. Jednou z prvních otázek, na něž si před řešením nějakého problému na počítači musíme odpovědět, je, zda z hlediska účelu a funkce programu má smysl sestavit jej celý v assembleru. Jsou oblasti, v nichž by jeho přímá aplikace hraničila s bláznovstvím - to se týká především tvorby matematických programů. Některými vyššími programovacími jazyky můžeme řešit složité matematické funkce neporovnatelně snadněji než assemblerem. Na větších počítačích dnes programuje v assembleru už jen málokdo, protože překladače vyšších jazyků na nich dokáží generovat velmi efektivní kód. Na druhou stranu, u mikropočítačových aplikací typu slovního procesoru, datových bází, práce s obrazovými daty a obecně u všech vstupně-výstupních operací bychom měli jednoznačně dát přednost assembleru právě pro jeho rychlost, která je u tohoto typu programů požadavkem nejvyšším.

A tak nám jako vždy zbývá zlatá střední cesta - jakýkoli vyšší programovací jazyk doplňovat rutinami strojového kódu, kdykoli se to ukáže být účelné. A naopak."

„Shrnu-li si to, pak dovedná, programově řízená manipulace s dispozicemi hardwaru počítače a jeho periférií spočívá na rychlých změnách dvou logických úrovní té které operace se právě účastnících prvků logických obvodů. A protože tyto změny provádí počítač velmi rychle, je i počet operací velmi vysoký."

„Ano. Počítač vlastně nic jiného 'neumí'. Proto mu, chudákovi, dal pan Howard Aiken nehezkou přezdívku ‚šíleně rychlý blbec na přesýpání jedniček a nul'. Přesněji by měl být označen jako nástroj umožňující programovatelnou simulaci procesů a dějů. Coby šíleně rychlý simulátor (nikoli simulant) je počítač bez konkurence."

„Aha, takže programování je vlastně i takové modelování. Ale přece jen - nemohli bychom už začít modelovat?"

„Když už to nemůžeš vydržet, zítra se podíváme na to, jak je strojový program uložen v paměti a jak je z něj odvozen assembler."

ZAKÓDOVANÝ PÁTEK

„AHOJ"

„41H48H4FH4AH"

„Cože? Nejsi trochu přepracován?"

„Nikoli, jen jsem ti odpověděl na pozdrav HD čísly mezinárodně přijaté tabulky znakových kódů ASCII. Tato čísla symbolizují znaky AHOJ. Když je uložíš do paměti počítače a budeš s nimi pracovat jako s daty reprezentujícími kód ASCII, budou znamenat právě slovo AHOJ."

„Ale s těmito kódy bych nápis AHOJ na našem domě přece nerozsvítil."

„Jak jsi poznal, každé písmeno nápisu AHOJ na naší obrazovce-domu bylo vyjádřeno 8 bajty (po 8 bitech). Dohodnuté kódy ASCII spotřebuji pro uložení 1 znaku v paměti jen 1 bajt. Ten má pro to které písmeno (znak) vždy stejnou, jemu odpovídající hodnotu."

„Takže pro zobrazení znaků na obrazovce nebo na tiskárně musím mít někde v paměti datovou tabulku výsledné sestavy jednotlivých bajtů jednotlivých znaků. Podle toho, jaký kód ASCII je zrovna interpretován, mi nějaká umná rutina vybere z tabulky dat patřičné bajty a sestaví z nich obrázek znaku. Znamená to snad, že vzhled znaků si vlastně mohu sestavit, jak budu chtít?"

„Vidíš, už jsi narazil na programátorský oříšek související s komunikací člověka s počítačem. Teď jsme si hráli s daty programu. Dál se podíváme, jak je to s instrukcemi strojového kódu.

Skutečnou podobou strojového kódu je binární kód. Pokud bys zvládl programování v tomto kódu, v podstatě by ses nemusel zabývat žádnou jeho výšejazyčnou odvozeninou. Jenže ... to bys musel mít v hlavě aspoň jeden mikroprocesor, nebo se vyznačovat schopností zapamatovat si obsah telefonního seznamu. Protože nic z toho zdrcující většina lidí nemá, bylo nutno přikročit při zápisu k takovým formám, které jsou bližší lidské, leč vzdálenější počítačové logice. Čím blíže je zápis lidské logice, tím rychleji programátor sestaví fungující program.

Ale za všechno se platí. V tomto případě je základní investicí čas. Čím jasnější řeč, hutnější forma zápisu, tím déle bude program data zpracovávat. Proto je takový rozdíl mezi rychlostí, s jakou pracuje program vytvořený ve strojovém kódu a rychlostí programu zapsaného třeba v jednoduché verzi Pascalu.

Podívej se, jak vypadá zápis kratičkého programu v binárním kódu. Tento prográmek sečítá dvě čísla z adres A160H a A161H a výsledek součtu ukládá na adresu A162H. Obsahuje 11 bajtů po osmi bitech:

00111010
01100000
10100001
01000111
00111010
01100001
10100001
10000000
00110010
01100010
10100001

Počítač by se nad takovou literaturou jen rozplýval, ty už asi o mnoho méně.
Výše uvedený binární zápis 11 bajtů nyní převedeme do formy hexadekadické:

3A
60
A1
47
3A
61
A1
80
32
62
A1

Žádné počtení ani pro počítač, ani pro člověka. Počítač pro vyluštění těchto čísel potřebuje pomocný program, který mu je pomůže převést do binárního kódu. Pokud jde o člověka, jen největší nadšenec programování ve strojovém kódu je schopen si zapamatovat HD tvar všech instrukcí Z80. A to mluvíme o osmibitovém mikroprocesoru, který má jen kolem 700 instrukcí. Ale představ si instrukční soubor takového 16-bitového nebo 32-bitového mikroprocesoru! Záplava HD kódů naroste do obrovských rozměrů a se schopnostmi své paměti jsme pak už zcela vedle. Proto byl pro programování ve strojovém kódu sestaven srozumitelný soubor assemblerových instrukcí, tzv. mnemonický kód.

Jistě si vzpomeneš, jak se ti učitelé snažili pomoci různými mnemotechnickými pomůckami při zapamatování nějaké složitosti. Třeba EKOKRPKA - pomůcka pro zapamatování si schematické struktury vývoje dramatu: Expozice, KOlize, KRize, Peripetie, KAtastrofa (vývojový diagram dramatu).

Dnešnímu písmu předcházelo písmo mnemonické. Bylo vlastně kódovaným sdělením pomocí zářezů do různých materiálů, uzlů na provázku apod. Jak vidíš, kódování holdovali už naši předkové. Mnemonika a mnemotechnika mají mnoho společného.

Dovedně sestavená mnemonika strojového kódu nám ve výsledku podává vyjádře-

ní všech instrukcí mikroprocesoru ve velmi účelné, snadno zapamatovatelné (srozumitelné) formě. Naučit se jí nazpaměť zabere méně času než šprtání syntaxe jakéhokoli vyššího programovacího jazyka.

Musíš si však opět uvědomit, že jestli jsme se vlastní řeči počítače vzdálili už HD kódem, pak mnemonika assembleru nás zavádí ještě dál. Nic strašného se však neděje, protože ji budeme používat jen při vlastním programování, tedy při zápisu instrukcí do editoru generátoru strojového kódu. Do něj vkládáme assemblerové instrukce, které editor ukládá do paměti ve formě zdrojového kódu (angl. *source code).* Generátor převede (zkompiluje) náš assemblerový program (jeho zdrojový kód) přímo do výsledného binárního (strojového) kódu, zvaného též cílový kód (angl. *object code).* V obou případech nejsme nijak zkráceni o možnost dodatečných úprav vytvářeného programu. Nyní se podíváme, jak bude vypadat zápis výše uvedených 11 bajtů v assembleru:

```
LD      A, (A160H)
LD      B,A
LD      A, (A161H)
ADD     A,B
LD      (A162H),A
```

Počítač si opět nepočte, ty však ano (i když možná až o něco později). Každopádně i absolutní laik pozná, že v tomto zápisu se ze zadané úlohy zřetelně objevují aspoň čísla adres. Dále je zde zřejmé, co jsou instrukce a co data, zatímco v předchozích dvou zápisech nebylo patrné ani tohle. To, že assemblerových instrukcí je jiný počet, než je reprezentujících 11 bajtů, je dáno tím, že jednotlivé instrukce obsahují od 1 do 4 bajtů.

Ukázky programů budeme zapisovat v této formě:

Adresa paměti	hexadek. kód	assembler	komentář
A150	3A60A1	LD A, (A160H)	;Přenos obsahu adr. A160H do reg. A
A153	47	LD B,A	;Přenos obsahu reg. A do reg. B
A154	3A61A1	LD A, (A161H)	;Přenos obsahu adr. A161H do reg. A
A157	80	ADD A, B	;Součet obsahu reg. A a reg. B
A158	3262A1	LD (A162H),A	;Uložení výsledku z reg. A na ;adresu A162H

Takovýto zápis poskytuje veškerou potřebnou orientaci v umístění instrukcí v paměti, umožňuje je do ní ukládat jako HD kód (s pomocí monitoru paměti) nebo jako assemblerové instrukce (s pomocí editoru/generátoru strojového kódu) a připojenými komentáři vysvětluje, jak program pracuje. Komentář je u každého assemblerového programu velmi cenný, protože bez něj se funkce, struktura a průběh programu zjišťují jen velmi obtížně.

V computerové literatuře celého světa je zápis mnemoniky mikroprocesoru Z80 shodný. Pochází přímo od jeho původního výrobce, firmy Zilog.

Čísla adres uložení programu nejsou nutně směrodatná. Teoreticky lze program do

volné paměti RAM umístit kamkoli libo. Je však třeba dát pozor na to, aby v případě výskytu instrukcí volání, přímých skoků atd. byla provedena odpovídající adresová, resp. datová transpozice. To je ovšem teorie. Pokud program sám o sobě není relokovatelný, umisťujeme jej v paměti tam, kam je předem stanoveno. Zdrojový text assembleru (díky jeho symbolickým adresám) nám však umožňuje umístit program relativně kamkoli. To platí i pro připojování dalších rutin ve tvaru zdrojového textu k základnímu programu."

„Už chápu, proč jsi mne předevčírem nechtěl pustit k programování. Co mě ještě čeká?"

„Před vlastním studiem instrukčního souboru Z80 musíš projít ještě dvěma vstupními branami softwarového paláce. Zítra zaklepeme na mikroprocesor."

MIKROPROCESOROVÁ SOBOTA

„Zatím vím něco jen o jednom konci adresové a datové sběrnice. Jak tuším, druhý bude právě v mikroprocesoru."

„Tyto sběrnice si můžeš představit jako součást krevního oběhu počítače. Cévy procházejí mikroprocesorem coby srdcem počítače. Často užívané přirovnání mikroprocesoru k mozku počítače je nevhodné, protože i ty funkce mozku, o nichž máme dnes nějakou představu, jsou funkcím dnes užívaných mikroprocesorů na hony vzdáleny. Když zavřeme obě oči nad „softwarovým vybavením" mozku, pak by jeho hrubě zjednodušenou analogií byla celá vnitřní struktura počítače, který by obsahoval nikoli jeden, ale celé kvantum mikroprocesorů. Takové počítače už dnes vystrkují růžky - říká se jim transputery, resp. multiprocesorové systémy, resp. počítače s paralelní logikou.

Mikroprocesor (podle jim dekódovaného programu) řídí průběh změn logických stavů buněk paměti, vstupně-výstupních portů i svých vlastních registrů a dalších obvodů své vnitřní struktury. Datová a adresová sběrnice nejsou jediné, po kterých běhají data při komunikaci s mikroprocesorem. On sám má i své vnitřní sběrnice. Podívej se na jeho blokový diagram v příloze.

Pro tebe, coby budoucího assemblujícího programátora, jsou nejdůležitější registry mikroprocesoru. V podstatě nejsou ničím jiným, než paměťovými buňkami uvnitř mikroprocesoru. Tyto buňky neadresujeme čísly, ale písmeny. Analogicky z nich opět odebíráme, nebo do nich ukládáme jednotlivé bity a bajty.

Pokud jde o odběr, resp. přenos dat, je tato terminologie trochu zavádějící. Přenášíme-li data odkudkoli (z adresy paměti či registru mikroprocesoru), pak na místě, z něhož jsou odebírána, zůstávají data nezměněna (jsou tam pouze přečtena). Odběr dat je 'sejmutím otisku' logického stavu bitů z místa odběru pomocí sběrnice, jejíž jednotlivé linky se tak dostanou do odpovídajících logických stavů. Změní se však logické stavy bitů adresáta, na něž jsou logické stavy z místa odběru přenášeny (do nichž jsou zapsány). Přenos dat je tedy kopírovacím procesem (s přihlédnutím k ev. přítomné inverzní logice).

Jednotlivé registry mikroprocesoru Z80 mají tyto názvy:

reg. A	Accumulator (akumulátor, střádač)
reg. F	Flags (praporky - stavové indikátory)
reg. BC	Byte Counter (bajtový čítač)
reg. DE	DEstination (určení - cílová adresa přenosu bajtů)
reg. HL	High, Low (vyšší a nižší část dvoubajtové adresy)
reg. IX	pro indexované adresování
reg. IY	pro indexované adresování
reg. SP	Stack Pointer (ukazatel zásobníku)
reg. PC	Program Counter (programový čítač - adresa kroku)
reg. R	Refresh (občerstvování dynamické paměti)
reg. I	Interrupt vector (vektor přerušení)

Všechny registry značené dvěma písmeny jsou 16-bitové. Registry BC, DE a HL, zvané **párové,** tvoři důležitou výjimku v tom, že kteroukoli jejich půlku lze používat i jako samostatný osmibitový registr. Registry značené jen jedním písmenem jsou jednobajtové. Názvy registrů jsou zvoleny velmi vtipně a adekvátně jejich poslání. Párové registry BC, DE, HL jsou však mnohem univerzálnější, než by napovídaly jejich názvy.

Registry A, F, B, C, D, E, H, L jsou v mikroprocesoru obsaženy dvakrát, ve dvou bankách, číslovaných 0 (hlavní) a 1 (vedlejší). Registry vedlejší banky se značí apostrofem u jejich názvu. K jejich aktivování je musíme programově ‚prohodit' s registry banky hlavní.

Zvláštní postavení mezi všemi mají registry A a F. K reg. A se vztahují některé logické a aritmetické operace, které nemůžeme provádět se žádným jiným registrem.

Zcela specifické funkce plní registr F a jeho **stavové indikátory** (zkráceně SI). Každý z nich je reprezentován jedním bitem, který může být programově testován, zda se nachází ve stavu logické 1 nebo logické 0. Výsledek takového testu je pak směrodatný pro další běh programu. Jde o test programem stanovených podmínek, po jejichž splnění či nesplněni se provedou programové skoky. Jako každý jiný registr, má i registr F osm bitů. V praxi je jich využito šest. Programátorovi jsou přístupné čtyři z nich (CY, Z, S, P/V). V každém programovém kroku (při realizaci jednoho příkazu) můžeme testovat vždy jen jeden z uvedených čtyř.

Registr R spolupracuje na občerstvování ‚unavených' kondenzátorků paměťových buněk dynamické paměti. V něm obsažené neustále se měnící číslo je kódem řádek paměťových matic, které jsou cyklicky občerstvovány. Díky své nestálosti se obsah registru R někdy programově využívá jako zdroj pseudonáhodných čísel. S registrem I se blíže seznámíš až v závěru učebnice.

Je-li počítač v chodu, ale zrovna ‚neprožívá' žádný vložený program, neznamená to, že nežije vůbec. V počítači (a samozřejmě i v mikroprocesoru) probíhá současně **řada precizně časovaných operací**, které jsou vnitřně synchronizovány kmitočtem zabudovaného oscilátoru. Synchronizace dějů v počítači probíhajících je jedním z nejdůležitějších procesů. Bez ní by jakákoli vnitřní či vnější komunikace nebyla myslitelná, nastal by stav totální dezorganizace a dezorientace. Mikroprocesor má 38 vstupních a výstupních linek, jimiž probíhají jednotlivé digitální pulsy a které slouží nejen pro přenos dat po da-

tové a adres po adresové sběrnici, ale upozorňují jej i na to, zda má v ten který moment číst (READ) nebo zapisovat (WRITE) bity a bajty, či čekat (WAIT) na programem definovaný vnitřní nebo vnější podnět atd., atd."

„I když sotva dýchám pod přívalem nových informací, přece jen se mi zdá, že registrů využitelných programátorem není zase tak moc."

„Máš pravdu. Většinou bys jich pro manipulaci s daty potřeboval několikrát tolik. To je jen další příčina toho, proč konstrukce assemblerových programů musí být velmi detailně promyšlena, strukturována. Zatímco v Basicu napíšeš PRINT „AHOJ" a o víc se nestaráš, v assembleru pro totéž musíš vymyslet několik funkčně provázaných rutin, jejichž tvorba ti zabere nejméně jeden den. A některé z registrů jsou ještě průběžně obsazeny operačním systémem počítače, takže je de facto můžeš odečíst."

„Jaký je takový hlavní funkční princip programu samotného?"

„Naprosto jednoduchý - manipulace s daty. Všechno, co do hotového programu vkládáš ke zpracování a na jeho výstupech odebíráš, jsou data. To, co provádí požadovanou manipulaci s daty, jsou instrukce, resp. programové příkazy. Ty jsou funkcemi programu; funkčně pracují s daty jako s konstantami a proměnnými programových funkcí. Program by se tak dal připodobnit k chytře postavené automatické lince, do jejíhož vstupu nasypeš hromadu plechu a na opačném konci vyjede hotový traktor, jehož jednotlivé díly nebo i fáze jeho výroby můžeš kdykoli i zpětně sledovat a do procesu zasahovat. Požadavek na program může ovšem být i zcela opačný - do vstupu vjede krásný traktor a na výstupu vyleze hromada plechu. Zdánlivě nesmyslné, ale podobný program může sloužit třeba pro hledání úspory materiálu. V posledním dni našeho seminárního týdne se k některým programovým otázkám ještě vrátíme."

NEDĚLE NA LINCE

„Pro pochopení dějů odehrávajících se v počítači je nutné poznat způsob jeho 'myšlení'. Alespoň pro dnešní den připusťme, že počítač má svůj způsob myšlení.

Počítač s jedním mikroprocesorem zpracovává všechny příkazy programu, který do něj vložíme, postupně. V každém okamžiku je schopen provádět jen jednu jedinou operaci, aniž by jakkoli chápal souvislosti nebo význam a obsah zpracovávaných dat. Myšlení počítače jde skutečně po jediné lince, kterou mu určujeme svým programem. Ale když tvrdíme, že je počítač tak rychlý, tak by přece měl projít *od* začátku do konce programu za chvilku, ne? Jistě, to by mohl. Jenže my mu do cesty stavíme něco, bez čeho by celá výpočetní technika rovněž nebyla myslitelná - podmínky, vlivem kterých počítač provádí cykly (opakování operací) a odskoky v programu.

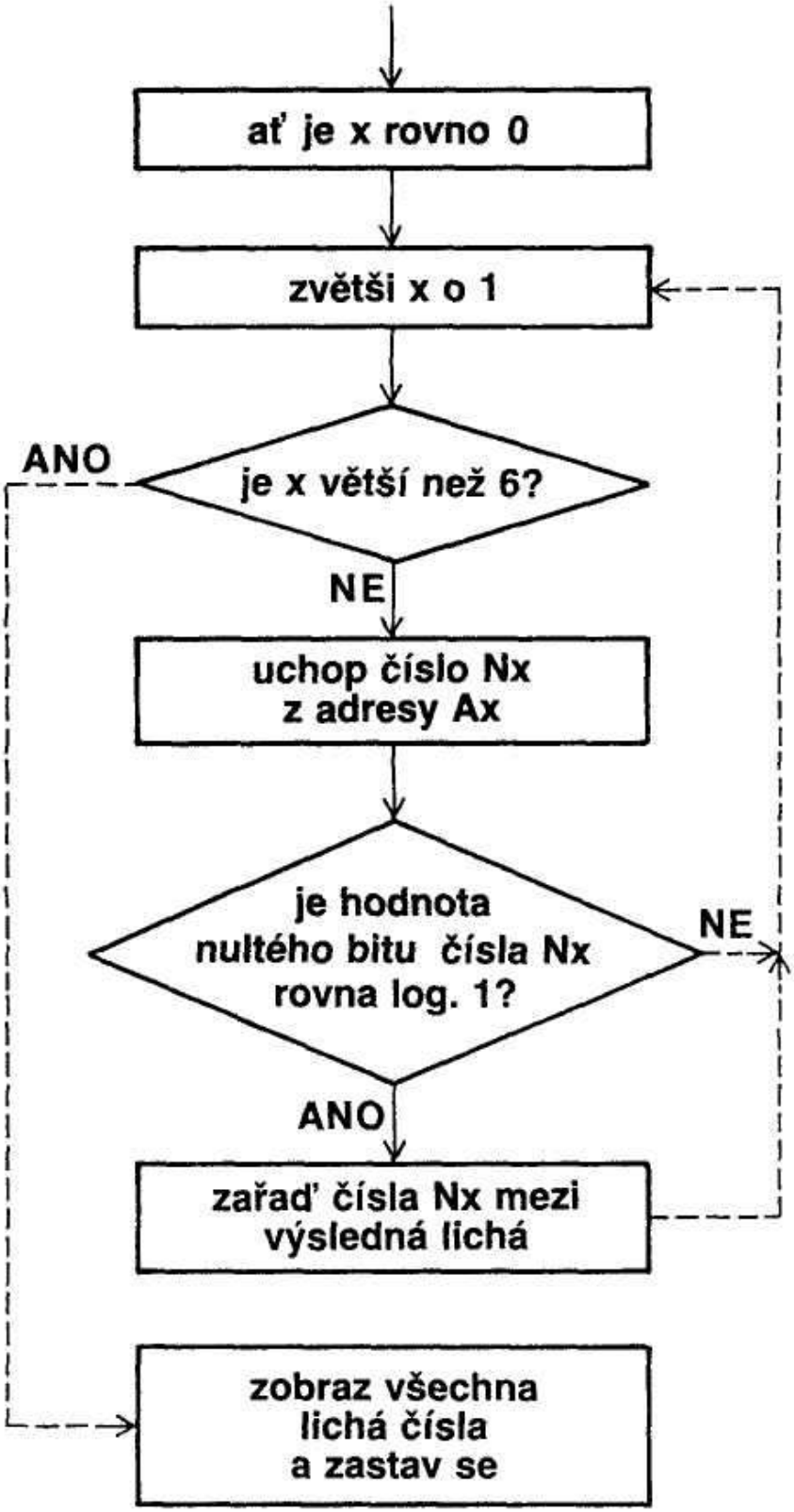
Jak třeba počítač vynásobí 3 krát 5? Prozaicky vyjádřeno musíme připravit program s tímto sledem operací:

1. Vynuluj určené místo (adresu A) v paměti.
2. Uchop do programu námi vložené číslo 5 a přičti je k obsahu adresy A (5 plus 0, tedy 1 krát 5).
3. Už jsi číslo 5 přičetl třikrát? POKUD NE (podmínka), pokračuj v přičítáni.
4. Přičti číslo 5 k obsahu adresy A (už tam bylo 5, teď to bude 10, tedy 2 krát 5).
5. Už jsi číslo 5 přičetl třikrát? POKUD NE, pokračuj v přičítání.
6. Přičti číslo 5 k obsahu adresy A (už tam bylo 10, teď to bude 15, tedy 3 krát 5).
7. Už jsi číslo 5 přičetl třikrát? POKUD NE, pokračuj v přičítání (na tuto podmínku teď počítač samozřejmě nereaguje); POKUD ANO, pípni, zablikej, zobraz na monitoru obsah adresy A a zastav se (čekej na případný další příkaz).

Počítač pípne, obrazovka zabliká a na ní se objeví číslo 15 (pokud je necháme zobrazit v soustavě dekadické). Kdybychom se podívali na bitovou prezentaci bajtu na adrese A, objevila by se nám už dobře známá bitová sestava 00001111. Jak z ‚nalinkovaného myšlení‘, které musíme pro sestaveni jakéhokoli programu plně respektovat, vyplývá, počítač neumí ani malou násobilku. Umí totiž jen ‚sečítat‘ elektrické potenciály na vstupech hradel, resp. počítat v hranicích jednoduché Booleovy algebry.

Schéma vývoje programu se samozřejmě nerozepisuje tak zeširoka. Nahrazuje se rozepsáním programových funkcí do vývojového diagramu (angl. *flow chart),* který je o něco přehlednější. V něm nejsou cykly opakování smyček rozepsány tolikrát, kolikrát se provedou, ale vyznačí se jen mezní podmínka takového opakováni.

Nakreslíme si jednoduchý vývojový diagram podprogramu, který bude vyhledávat lichá čísla v intervalu od 0 do 255 z řady např. šesti čísel Nx (kde x postupně nabývá hodnoty 1 až 6), uložených na sousedících adresách Ax:



Vložíme-li ke zkoumání třeba tuto řadu šesti čísel - 22,14, 53, 2, 64,123 — na obrazovce se objeví všechna lichá čísla této řady, tedy 53 a 123, protože jejich nultý bit má hodnotu logické 1.

Povšimni si mluvy diagramu. Není ti nápadné, že zkoumá jen to, zda je něco větší nebo menší či rovno čemusi? Nebo k něčemu něco přičítá či tomu přiřazuje hodnotu, něco někam ukládá a zase to odebírá ... Rozhodně žádná velká květomluva. I když mikroprocesor má kolem 700 instrukci, vlastních typů operací je mnohem méně. Některé skupiny instrukcí jsou jen variacemi na dané téma.

Všimni si dvou míst, v nichž jsou rozhodovací body. Z nich vedou vždy dvě cesty. O tom, kterou se mikroprocesor vydá, rozhoduje výsledek programově zadaného testu. Je-li podmínka testu splněna, mikroprocesor nebude pokračovat přímo za nosem (tedy na nejbližší instrukci směrem k vyšším adresám), ale převede programové řízení na adresu, která je přímo či nepřímo určena podmíněnou instrukcí. Jak už víš, tato adresa se musí objevit v jeho programovém čítači.

Kdo rozhoduje o tom, kterou ze dvou cest vedoucích z rozhodovacích bodů se mikroprocesor vydá, je svědomitý výhybkář sídlící v registru F. K ruce má skvělé pomocníky, o nichž jsme si už něco řekli - stavové indikátory (budu je dále zkracovat na tvar SI). Tak např. SI nuly (značí se Z podle angl. slova *Zero,* česky nula). Když do programové linky zařadím podmíněnou instrukci JP Z,1234, v momentě, kdy ji bude mikroprocesor dekódovat, proběhne jeden ze dvou možných dějů:

* Je-li SI nuly ve stavu logické 0, Z=0 (operace provedená předchozí jej ovlivňující instrukcí měla nenulový výsledek), pak mikroprocesor bude nerušené pokračovat dekódováním instrukce uložené ihned za JP Z,1234.
* Je-li SI nuly ve stavu logické 1, *Z* = 1 (operace provedená předchozí jej ovlivňující instrukcí měla nulový výsledek), pak bude do programového čítače zapsána adresa 1234. Mikroprocesor ‚skočí' na tuto adresu a bude dekódovat na ni (resp. na dalších adresách) uloženou instrukci.

Programátorovi je zpřístupněno zjišťování logických stavů čtyř SI. Každý z nich může být ve stavu logické 0 nebo logické 1. Zdá se to málo, ale ve vlastní praxi to bohatě stačí. Když se podíváš na list přílohy, v němž je uvedeno, jak které instrukce ovlivňují které indikátory, zjistíš, že některé je neovlivňují vůbec, většina zbývajících pak jen některé z nich. Proto je třeba dbát nejen na výběr instrukce pro zjišťováni stavu toho kterého indikátoru, ale také správně zvolit typ instrukce, která je schopna ten který indikátor předem plánovaně ovlivnit.

Využití řečeného si ukážeme na programové smyčce (angl. *loop,* čti lůp). Naše smyčka bude mít za úkol provést 65535 cyklů. Po jejím ukončení bude programové řízení převedeno na jinou adresu. Strategie tvorby smyčky je následující:

Napřed musíme někam uložit dvoubajtové číslo 65535 - samozřejmě do nějakého párového registru. Zvolíme registr BC. Od něj budeme postupně odečítat vždy 1, dokud jeho obsah nebude nulový - tak obdržíme 65535 průchodů (cyklů). Každý cyklus bude začínat na adrese symbolizované návěštím ZNOVA (proto je assembler označován za jazyk symbolických adres). A nakonec musíme zjistit, kdy bude obsah registru BC nulo-

vý, abychom smyčku ukončili. Pokud bychom test nezařadili, dostali bychom se do tzv. věčné smyčky, z níž není lehko uniknout. Proto nesmíme moment vynulování registru BC propást. K číhání na tento okamžik použijeme Booleovu algebru. Smyčka bude vypadat následovně:

```
        LD BC, 65535    ;Ulož do reg. BC číslo 65535
ZNOVA:  DEC BC          ;Sniž obsah reg. BC o 1
        LD A, B         ;Ulož do reg. A obsah reg. B
        OR C            ;Proveď log. operaci OR mezi reg. A a C
        JP Z, KONEC     ;Je-li výsledkem 0, skoč na adr. KONEC
        JP ZNOVA        ;Není-li výsledkem 0, zpět na ZNOVA
```

Vezmi si k ruce pravdivostní tabulky logických operací. U logické funkce OR vidíš, že pouze při účasti dvou nulových bitů je výsledkem logická 0 (jinak vždy logická 1). V programu se logická operace provádí mezi těmi dvěma bity, které v obou bajtech mají stejné pořadové číslo. Co to znamená pro náš program? Že po přenosu obsahu registru B do registru A a následné operaci OR registru A s registrem C bude výsledkem nula tehdy a jen tehdy, budou-li všechny bity obou registrů ve stavu logické 0. A právě tehdy bude SI nuly Z=1. Instrukce JP Z, KONEC říká: Když je Z na úrovni logické 1 (tedy výsledek operace OR je nulový), pak si, milý mikroprocesore, dej do programového čítače adresu KONEC a skoč na ni (vyskoč z téhle smyčky). Když Z=0, nevšímej si mne a vykonej instrukci následující - zde JP ZNOVA (ulož do programového čítače adresu ZNOVA a skoč na ni, tedy proveď další cyklus smyčky)."

„Proč přenášíš obsah registru B do registru A, když by přece bylo lepší operovat rovnou s bity registrů B a C?"

„Protože mikroprocesor operaci B OR C nezná, umí provádět logické operace jen za účasti registru A. Podobných omezení, která Z80 řadí k tzv. nesymetrickým procesorům, je v jeho instrukčním souboru víc, postupně se s nimi seznámíš.

Doba, kterou zabere vykonávání této smyčky, je přímo úměrná číslu vloženému do registru BC. Jeho změnou tak můžeme regulovat dobu trvání celé smyčky. Toho se využívá všude, kde vzniká potřeba přesného časování průběhu operaci. Takovým smyčkám se pak říká časovací smyčky. To je však jen jedna z možností jejich využití. Vzpomeň si na náš nápis AHOJ. Každé písmenko se skládá z osmi bajtů. Pro konstrukci každého z nich budeme tedy potřebovat smyčku s osmi cykly, během nichž se postupně na určité adresy obrazové paměti uloží patřičných osm bajtů písmene či znaku... Ale copak, že jsi dnes nějak zticha?"

„Víc už mi ani neříkej. Je mi jasné, že teď se musím pustit do studia vlastních instrukcí. Podmínky a smyčky... nikdy bych byl neřekl, že na nich stojí cely program "

„Jsi skutečnosti blízko, ale nezapomeň na to, co je na programování skutečně tvůrčí - hledání vhodných algoritmů a jejich převodů na posloupnost programových operací. Podmínky i smyčky umožňují především strukturovat program tak, aby každý algoritmus byl prováděn v průběhu programu tehdy a tak, kdy a jak prováděn být má. Strukturu programu tvoři soubor podprogramů, jejichž skladba a vzájemná funkční (dějová) provázanost dává výslednou programovou funkci. Pak už zbývá jen do takového programu

vkládat vstupní data a kochat se tím, jak dovedně je programátorem vytvořené algoritmy zpracovávají.

Prošel jsi poslední vstupní branou softwarového paláce. Dále jsem pro tebe připravil pár desítek stran „samoučného" materiálu, který tě detailně seznámí s celým instrukčním souborem mikroprocesoru Z80. A protože se mi jedné krásné noci zdálo, že vyšel v nakladatelství Mladá fronta, nebudeš se snad zlobit, že v něm preventivně od tykání přecházím do oslovení v množném čísle."

„Jak dlouho, myslíš, že mi bude trvat vstřebání toho všeho?"

„Když se budeš studiu věnovat intenzívně, tak asi dva měsíce. Pokud bys to ale nezvládl do půl roku, bude lepší, když se poohlédneš po jiném koníčku."

„Za dva měsíce mě tu máš zpátky! 41H 48H 4FH 4AH!"

2. ČÁST

INSTRUKČNÍ SOUBOR MIKROPROCESORU Z80

KAPITOLA 1
ADRESOVACÍ MÓDY Z80
1. ČÁST

V dalším textu jsou zavedeny tři zkratky:
HD pro výraz hexadekadický
SI pro stavové indikátory
DK pro dvojkový komplement (doplněk)

INSTRUKČNÍ SOUBOR Z80

Než začnete vstřebávat jednotlivé instrukce tohoto souboru, bude vhodné si jej hned zkraje rozdělit na několik podsouborů z hlediska jejich funkce:

1. Instrukce přenosu dat

Budeme-li pojímat i registry Z80 a vstupně/výstupní porty počítače jako místa paměti, pak lze říci, že přenos dat je možný mezi jakýmikoli místy paměti s těmito hlavními omezeními:

- Z paměti ROM můžeme data jen odebírat (číst).
- Instrukční soubor neobsahuje instrukce vzájemného přenosu mezi některými párovými registry i jimi adresovanými místy paměti (v takovém případě je přenos nutno provést přes některý volný registr).
- Přenos může být jedno- či dvoubajtový, a to přímý, nepřímý nebo indexovaný. Tyto instrukce (značené LD) neovlivňují žádné SI s výjimkou dvou instrukcí — LD A, I a LD A, R.

2. Aritmetické a logické instrukce

V podstatě jde o instrukce přenosu, obohacené o aritmetické a logické operace s daty. Tyto operace probíhají v aritmeticko-logické jednotce (ALU) mikroprocesoru. Jednobajtové operace (log. a aritm.) se váži k reg. A (do něj, s výjimkou operace porovnávání, se ukládá i jejich výsledek). Dvoubajtové (jen součet a odečet) se váží k párovému reg. HL nebo indexovým reg. IX a IY. Typy logických instrukcí jsou: AND, OR, XOR, CPL, NEG; aritmetických: INC, DEC, ADD, SUB, CP, DAA. Tyto instrukce ovlivňují SI.

3. Instrukce pro změnu programového řízení

Není-li učiněn zásah do programového řízení, mikroprocesor interpretuje postupně jednu instrukci za druhou směrem k vyšším adresám. Adresa posloupné instrukce, která má potencionálně být vykonána v příštím programovém kroku, se vždy objeví v programovém čítači (reg. PC) Z80. Když do programového čítače uložíme neposloupnou adresu, převedeme na ni i programové řízení. Takto měnit obsah reg. PC mohou instrukce skoku (JP, JR), volání (CALL, RST) a návratu (RET). Tyto instrukce neovlivňují SI. Mohou být i podmíněné - pak se provedou tehdy, je-li v nich obsažená podmínka (zjišťovaný stav určitého SI) splněna. Měnit programové řízení můžeme také při práci s jednotlivými typy a módy přerušení.

4. Instrukce bitových manipulací

Velkou část instrukčního souboru Z80 tvoří instrukce typu SET, RES, BIT. Pomocí nich lze změnit stav kteréhokoli bitu většiny registrů a paměti RAM a testovat stav kteréhokoli bitu těchto registrů a paměti RAM i ROM. Z nich ovlivňují SI pouze instrukce testující. V reg. F můžeme přímo ovlivňovat jen jeden jeho bit - SI Carry. Do této skupiny dále patři instrukce pro rotaci a posuv bitů jednoho bajtu adresy nebo registru a rotaci ' skupin 4 bitů mezi reg. A a místem v paměti. Tyto instrukce ovlivňují SI.

Pozn.: Obecně vzato by do této skupiny patřily všechny instrukce, které ovlivňují stav jakéhokoli bitu.

5. Instrukce vstupně/výstupní

Přenos dat mezi počítačem a vnějším zařízením prostřednictvím adresovaných vstupně/výstupních portů. SI jsou ovlivňovány různou měrou.

6. Blokové instrukce

Týkají se přenosu (souvztažnost k bodu 1) a prohledávání (souvztažnost k bodu 2) bloku dat v paměti počítače i přenosu dat mezi počítačem a vnějším zařízením (souvztažnost k bodu 5). Parametry přenosu i prohledávání jsou předem definovatelné. SI jsou ovlivňovány různou měrou.

7. Instrukce přerušení

Zastavení činnosti mikroprocesoru, blokování a uvolnění maskovatelného přerušení a stanovení jeho módu. Tyto instrukce nemají vliv na SI.

8. Instrukce NOP (No Operation)

V překladu „žádná operace". Její kód je nula. Bez vlivu na SI.

Instrukční soubor Z80 zahrnuje 10 typů adresovacích módů, které určují, odkud kam budou přenášena data nebo odkud kam bude převedeno programové řízeni. Obě místa 'odkud' i 'kam' musejí být přesně určena, adresována. Proto hovoříme o adresovacích módech.

V této kapitole začneme těmi nejméně komplikovanými přenosy dat. Z přílohy učebnice si vezměte k ruce soupisy instrukcí Z80. Hledejte v nich i jiné instrukce toho kterého typu přenosu než v příkladech uvedené. Všímejte si, kolik bajtů a jaké obsahuje jejich strojový kód. Brzy tak odhalíte četné souvislosti. Povšimněte si i toho, že v mnoha případech lze provádět určitý typ přenosu jen s určitým (-i) registrem (-y). Instrukční soubor Z80 tedy nevyčerpává všechny teoreticky možné přenosy mezi registry i adresami. To je jeden z důvodů, proč v prvních měsících (často i letech) vašeho samostatného programování bude pro vás soupis instrukcí neocenitelnou pomůckou.

REGISTROVÉ ADRESOVÁNÍ

Jednobajtový přenos obsahu registru

V instrukčním souboru Z80 je 63 různých instrukcí tohoto typu. Každá z nich má mnemoniku LD u, z kde:

u - registr určení
z - zdrojový registr

Příklady instrukcí:		**HD kód:**
LD C, B	přenos obsahu registru B do registru C	48
LD A, C	přenos obsahu registru C do registru A	79
LD D, E	přenos obsahu registru E do registru D	53
LD A, H	přenos obsahu registru H do registru A	7C
LD L, A	přenos obsahu registru A do registru L	6F

Dvoubajtový přenos obsahu 16-bitového registru

Z80 zná jen 3 takové instrukce:

LD SP, HL	přenos obsahu registru HL do registru SP	F9
LD SP, IX	přenos obsahu registru IX do registru SP	DD F9
LD SP, IY	přenos obsahu registru IY do registru SP	FD F9

Je nutno mít na paměti, že po přenosu obsah registru zdrojového zůstává nezměněn, zatímco obsah registru určení se změní (jeho obsah se stane kopií obsahu zdrojového registru). Registry účastnící se na provedeni instrukce jsou přímo obsaženy v jejím operačním kódu.

BEZPROSTŘEDNÍ ADRESOVÁNÍ

Přímý přenos jednobajtových dat do registru (adresy paměti)
LD r, NN; LD (HL), NN

Mnemonika tohoto přenosu je LD r,NN, přičemž r je registr určení. U instrukce LD (HL),NN jde o adresu určení, která je v tomto případě adresována nepřímo (viz níže). NN je číslo v rozsahu 00H-FFH a je přímou součástí instrukce.

Příklady instrukcí:	**LD B, 12H**	**LD (HL),** F1H
a jejich HD kód:	**06 12**	**36 F1**

BEZPROSTŘEDNÍ ROZŠÍŘENÉ ADRESOVÁNÍ

Přímý dvoubajtový přenos dat do párových registrů
LD rr, NNNN

Tyto instrukce patří k tzv. 16-bitovým, protože přenášejí dva bajty, tedy 16 bitů (NNNN je v rozsahu 0-FFFFH) do jednoho z těchto párových registrů: BC, DE, HL nebo SP. Tento přenos se nazývá bezprostřední proto, že tu jde o okamžitý, ne zprostředkovaný přenos (data, podobně jako v předchozím případě, jsou přímou součástí instrukce).

Příklady instrukcí:	LD BC, 1234H	LD SP, ABCDH
a jejich HD **kód:**	01 34 12	31 CD AB

Zde se poprvé setkáváme se základním pravidlem syntaxe instrukčního souboru Z80, v němž zásadně platí, že pokud za operačním kódem následuji dva datové bajty (vždy se jedná o číselnou reprezentaci adresy), do paměti se za operační kód ukládá napřed nižší bajt a teprve za něj vyšší bajt dvoubajtového čísla (adresy).

REGISTROVÉ NEPŘÍMÉ ADRESOVÁNÍ

Nepřímý jednobajtový přenos s registrem
LD r,(rr); LD (rr),r

V dosud uvedených módech byla ukládaná (přenášená) data vždy obsažena přímo v instrukci. V tomto módu operujeme s datovými bajty, které nejsou součástí instrukce, ale jsou adresovány registry v instrukci uvedenými (určují, na kterou adresu data uložit, nebo ze které adresy, resp. kterého registru data odebrat).

Párový registr BC, DE nebo HL obsahuje adresu paměti (v rozsahu 0000H-FFFFH), z níž nebo na niž se uskutečňuje přenos za účasti reg. A. Tak např. LD A,(DE) zname-

ná, že obsah adresy (jeden bajt) určené číslem v párovém registru DE je přenesen (uložen) do registru A. Naopak LD (BC),A způsobí přenos obsahu registru A na adresu paměti určenou obsahem párového registru BC.

Příklady instrukcí:

LD (HL), A přenos obsahu registru A do adresy určené obsahem registru HL (HD kód 77)
LD A, (HL) přenos obsahu adresy určené obsahem registru HL do registru A (HD kód 7E)

ROZŠÍŘENÉ ADRESOVÁNÍ

Instrukce tohoto módu obsahují ve svých posledních dvou bajtech přímo číslo adresy. Přenos dat může být jedno- nebo dvoubajtový.

Rozšířený jednobajtový přenos mezi akumulátorem a pamětí LD A,(adr); LD (adr),A

Tak např. instrukce LD A,(1234H) přenese obsah adresy 1234H do reg. A, instrukce LD (ABCDH), A přenese obsah reg. A na adresu ABCDH.

Rozšířený dvoubajtový přenos mezi párovými registry a pamětí LD rr, (adr); LD (adr), rr

V obou případech opět platí pravidlo zápisu dvou adresových bajtů do paměti ve stanoveném pořadí. Dejme tomu, že do instrukce potřebujeme zapsat adresu 65534 (dekadicky), což je FFFEH. Vyšším bajtem adresy se zde rozumí její horní půlka, tedy FFH, nižším dolní půlka, tedy FEH. Budeme-li chtít provést dvoubajtový přenos mezi adresou FFFEH a párovým registrem BC (mnemonika je LD (FFFEH), BC), pak HD tvar zápisu instrukce bude ED 43 FE FF s tímto významem:

ED 43 — operační kód instrukce
FE — nižší bajt adresy
FF — vyšší bajt adresy

Z toho je patrno, že assembler dodržuje běžnou logiku sledu zápisu čísel, zatímco HD prezentace instrukce se podřizuje pořadí, v jakém bajty musejí být uloženy přímo do paměti. Na to nikdy nezapomeňte. Omyly končívají ‚anihilací' programu.

Uvedený dvoubajtový přenos si můžeme teoreticky osvětlit pomocí jednobajtového. Např. instrukce LD (adr), HL proběhne v tomto sledu: LD (adr), L a LD (adr + 1), H. Podobně v případě LD HL,(adr) bude sled takovýto: LD L,(adr) a LD H,(adr + 1). Z toho je zřejmé, že i když je v zápisu instrukce uvedena jen jedna adresa, instrukce pracuje

i s adresou o 1 vyšší, přičemž přenos probíhá opět v předepsaném sledu - nejdříve je přenášen nižší bajt (na nižší adresu nebo z ní) a pak teprve bajt vyšší (na vyšší adresu nebo z ní). Pokud bychom chtěli např. instrukci LD HL,(1234H) přepsat do sledu instrukcí, které Z80 zná a s nimiž jste se dosud seznámili, mohl by jejich sled vypadat takto:

```
LD A, L                 nebo i takto:   LD BC, 1234H
LD (1234H), A                           LD DE, 1235H
LD A,H                                  LD A, L
LD (1235H), A                           LD (BC), A
                                        LD A, H
                                        LD (DE), A
```

K dosud uvedeným adresovacím módům přenosu dat patří i řada dalších instrukcí, které nejsou typu LD. Jejich funkce spočívá právě v provedení přenosu dat, někdy vícenásobném. O tom, jaké přenosy provádějí jednotlivé typy instrukcí, se budete dozvídat v průběhu jejich výkladu. Např. k adresování LD rr,NNNN náleží instrukce typu JP, k posledně uvedenému adresovacímu módu náležejí např. instrukce typu PUSH a POP.

Ze zápisu instrukcí přenosu dat vyplývá jedna zákonitost assemblerové syntaxe. Pokud je 16-bitový registr nebo absolutní adresa (ta je rovněž 16-bitová) zapsán v závorkách, instrukce operuje s OBSAHEM jí adresované adresy. Instrukce čte z adresy paměti nebo do ní zapisuje!

Pokud 16-bitový číselný parametr nebo registr téhož rozsahu nejsou uvedeny v závorkách, jde o přímý, bezprostřední přenos dat do registru - týká se prvních tří módů. Tyto instrukce s adresami paměti žádnou operaci neprovádějí.

V této syntaxi je bohužel jedna 'díra'. Týká se skokové instrukce JP. Jak už víte, jde o instrukci, která může změnit programové řízení. Např. JP 2345H převede řízení (provede skok) na adresu 2345H, protože uvedený číselný parametr, označující skokovou adresu (říká se jí také absolutní), je zapsán do registru PC. Proběhne tedy operace (zápis jen teoretický) LD PC,2345H. Užijeme-li místo absolutní adresy dvoubajtové registrové adresování, pak skoková instrukce může mít např. tento skutečný tvar: JP (HL). Zde proběhne takovýto přenos (zápis teoretický): LD PC,HL. I když v instrukci JP (HL) je HL v závorkách, instrukce neoperuje se žádnou adresou paměti. To platí i pro obdobné instrukce JP (IX) a JP (IY).

Na tomto místě je načase seznámit se s tím, co znamená doba provedeni jedné instrukce a proč se v tabulkách uvádí. Čas potřebný pro provedeni instrukce je přímo závislý na frekvenci, v níž pracuje mikroprocesor počítače. Z80 se vyskytuje v několika provedeních; podle typu (i schopností výrobce) může pracovat v kmitočtu 2-8 MHz. Firma HITACHI došla dokonce až k 10 MHz. Jak si v tabulkách můžete povšimnout, mnohé instrukce mají shodný počet taktů svého provedení. Je to dáno časovými průběhy operací, které v mikroprocesoru neustále probíhají. Instrukce LD r,NN vyžaduje 7 taktů, což pří 4 MHz odpovídá času 2,8 mikrosekundy. Všechny registrové i datové přenosy Z80 probíhají paralelně -všech 8 bitů je převáděno současně po 8 linkách. Určité speciální podmínky vyžaduje pouze nepřímý přenos dat při zápisu do paměti nebo čtení z ní.

Teď se chvíli přestaneme zabývat adresovacími módy a rozšíříme si naše znalosti o několik instrukcí, které použijeme v následujících experimentech.

Blokový přenos dat
LDD, LDI, LDDR, LDIR

Tato bloková instrukce sice není v základním rozdělení instrukcí uvedena v podskupině 1., ale mezi instrukcemi přenosu dat ji nelze opomenout.

Dosud jsme probrali několik způsobů přenosu dat mezi registry a adresami paměti v rozsahu 1-2 bajtů. Z80 má čtyři velmi užitečné funkce, které dokáží přenášet z jedné části paměti do druhé celý blok dat najednou (obě oblasti paměti se mohou i prolínat).

Před provedením této instrukce je nutno inicializovat (nastavit) obsah párových registrů BC, DE a HL takto:

HL — adresa prvního zdrojového bajtu
DE — adresa prvního bajtu určení
BC — počet přenášených bajtů (čítač jejich počtu)

Provedeni instrukce LDI *(Load-Increment)* vyvolá následující pochody:

1. Bajt v paměťové adrese určené obsahem reg. HL je přenesen na adresu určenou obsahem párového registru DE.
2. Obsahy registrů HL i DE se automaticky zvýší o 1.
3. Obsah reg. BC se sníží o 1.

Provedení instrukce LDIR *(Load-Increment-Repeat)* má týž účinek jako instrukce LDI, ovšem s tím velmi důležitým rozdílem, že se najednou provede přesun takového počtu bajtů, který je roven číslu obsaženému v párovém registru BC:

1. Bajt z paměťové adresy určené obsahem registru HL je přenesen na adresu určenou obsahem registru DE.
2. Obsahy registrů HL i DE se automaticky zvýší o 1.
3. Obsah registru BC se sníží o 1.
4. Obsah registru BC je testován. Když není nulový, pak se kroky 1, 2, 3 a 4 zopakuji. Když je nulový, provádění instrukce se zastaví a program pokračuje následující programovou instrukcí.

Anglické slovo *repeat* znamená opakuj.

Instrukce LDD *(Load-Decrement)* a LDDR *(Load-Decrement-Repeat)* jsou shodné se svými výše uvedenými dvojčaty, ale s tím rozdílem, že během provádění kroku 2 se v obou případech obsahy registrů HL i DE snižují o 1.

U instrukcí s opakováním můžeme tedy přenést nejvíce 10000H bajtů (65536 dekadicky) při počátečním obsahu reg. BC= 0.

Pro snadnější pochopení funkce těchto instrukcí si text uvedený u instrukce LDIR porovnejte se schématem z přílohy.

Kořením vašeho studia assembleru by měla být praktická demonstrace a analýza programů dále uváděných jako experimenty. Aby byly hned zpočátku zážívnější, skočíme na chvilku trochu dopředu a k instrukcím přenosu přidáme ještě čtyři další - INC a DEC z podskupiny aritmetických instrukcí a JP NZ, XXXX a RET pro změnu programového řízení.

Zvýšení obsahu registru o 1
INC r

Anglická zkratka INC znamená *increment (zvyš).* Po provedení této instrukce se obsah registru r zvýší o 1. Tak např. je-li obsahem registru B číslo 16, po instrukcí INC B bude jeho obsah 17.

Snížení obsahu registru o 1
DEC r

Opak instrukce INC r. Po jejím provedení se obsah registru r sníží o 1. V souvislosti s těmito instrukcemi nezapomeňte, že je-li obsah registru před provedením instrukce INC r roven FFH, po jejím provedení bude obsahovat 0. Provedete-li ihned nato instrukci DEC r s tímtéž registrem, bude jeho obsah FFH.

Přímý skok s podmínkou NOT ZERO JP NZ, XXXX

Je-li při provádění této podmíněné skokové instrukce SI nuly ve stavu log.0, programové řízení přejde ihned na adresu XXXX (provede se skok na tuto adresu). Pokud výsledek předchozí operace, ovlivňující SI nuly, bude nulový (Z= 1), pak se skok neprovede a program bude pokračovat instrukci následující. Sama instrukce je tříbajtová, s jednobajtovým operačním kódem. Obsah adresy XXXX je obsahem 2. a 3. bajtu instrukce. Opět nesmíme zapomenout, že při jejím HD zápisu musí nižší bajt adresy předcházet vyšší.

K podrobnějšímu výkladu instrukce RET *(RETurn* - návrat) se dostaneme dále. Pro tuto chvíli si pamatujte, že nás *vrátí* tam, odkud jsme volali sled instrukcí umístěných před ní. Je tu tedy jistá analogie s basicovým příkazem RETURN.

Následujících šest experimentů demonstruje, co jste se naučili v této kapitole. Týkají se přenosu dat. Lze jen doporučit, abyste si zápisy jednotlivých experimentů na svém počítači provedli. Funkci každého programu tak pochopíte mnohem snadněji, i když nad věcí strávíte trochu více času. Je to však investice, která se vám bohatě vyplatí. 1 z toho důvodu jsou následující programové ukázky nazvány experimenty, nikoli jen prostými příklady k přelétnutí očima.

EXPERIMENT č. 1

```
A100  0680    LD B, 80H     ;Datový bajt 80H je uložen do reg. B
A102  04      INC B         ;Zvýšení obsahu reg. B o 1
A103  48      LD C,B        ;Obsah reg. B je přenesen do reg. C
A104  0C      INC C         ;Zvýšení obsahu reg. C o 1
A105  51      LD D,C        ;Obsah reg. C do reg. D ,
A106  14      INC D         ;Zvýšení obsahu reg. D o 1
A107  5A      LD E, D       ;Obsah reg. D do reg. E
A108  10      DEC E         ;Snížení obsahu reg. E o 1.
A109  63      LD H, E       ;Obsah reg. E do reg. H
A10A  25      DEC H         ;Snížení obsahu reg. H o 1
A10B  6C      LD L, H       ;Obsah reg. H do reg. L
A10C  2D      DEC L         ;Snížení obsahu reg. L o 1
A10D  7D      LD A, L       ;Obsah reg. L do reg. A
A10E  C9      RET           ;Návrat k adrese volání
```

KROK 1. Zapište uvedený program do paměti od libovolné adresy. Zkontrolujte si, jestli je program zapsán dobře. Protože jste se dali na cestu programátora, musíte si zvyknout na preciznost - odpovězte na tyto kontrolní otázky.

Kolik jednobajtových instrukcí je v programu? A dvoubajtových, tříbajtových a čtyřbajtových?

Pokud jste ještě neodpověděli, nedívejte se na správnou odpověď, nechcete-li podvádět sami sebe. Odpověď zní 13, 1, 0, 0. První instrukce programu je jediná dvoubajtové. Všechny ostatní jsou jednobajtové.

Kolik instrukci používá přenos jednobajtových dat? A kolik registrový přenos?

Správná odpověď - 1 a 6. instrukce LD B,80H je jediná, která přenáší data. Ostatní instrukce typu LD přenášejí obsah jednoho registru do druhého.

KROK 2: Analyzujte program (nejdříve s tužkou v ruce a potom třeba krokováním) a určete, jaký bude HD obsah registrů B,C,D,E,H,L,A po skončeni programu.

KROK 3: Pokud jste programem krokovali, určitě jste si všimli všech změn obsahů registrů programem ovlivněných. Správná odpověď na předchozí otázku je:

A = 80H, B = 81H,C = 82H, 0 = 83H, E = 82H, H = 81H, L = 80H

Umět předvídat změny obsahu registrů i jiné změny programu ve strojovém kódu je velmi důležité pro úspěšné odladění jakéhokoli programu. Vřele vám doporučuji, abyste si tuto schopnost vypěstovali. Bez tréninku ji však nezískáte.

KROK 4: Změňte datový bajt na adrese A101H na 01. A zkuste opět předpovědět konečné obsahy výše uvedených registrů. Správná odpověď:

A = 01, B = 02, C = 03, D= 04, E = 03, H = 02, L = 01

KROK 5: Změňte datový bajt na adrese A101H na FFH. Opět se pokuste určit konečný obsah registrů. Správná odpověď:

A = FFH, B= 00, C = 01, D=02, E = 01, H = 00, L=FFH

Tím se potvrzuje, co je uvedeno v odstavci věnovaném instrukci DEC r. Když zvýšíme FFH o 1, dostaneme 00H. Když snížíme 00H o 1, dostaneme FFH.

EXPERIMENT č. 2

Demonstrace dvoubajtového rozšířeného a nepřímého registrového adresovacího módu.

```
A1102 11CA1      LD HL, A11CH     ;A11CH do reg. HL; A1 do H; 1C do L
A113  36FF       LD(HL),FFH       ;FFH na adresu A11CH
A115  2C         INC L            ;Zvýšení obsahu reg. L o 1. Tak je
                                  ;v reg. HL A11DH
A116  36EE       LD(HL), EEH      ;EEH na adr. A11DH
A118  2A1CA1     LD HL, (A11CH)   ;Rozšířený adres. mód. Bajt z adresy
                                  ;A11CH do reg. L, z A11DH do reg. H
A11B  C9         RET              ;Návrat k adrese voláni
```

KROK 1: Zapište program a zkontrolujte správnost zápisu v počítači.

KROK 2: Celý program pracuje pouze s párovým reg. HL a dvěma adresami A11CH a A11DH, na něž ukládá bajty s obsahem FFH a EEH. Poté převede obsahy těchto dvou adres do párového registru HL. Nejde zrovna o program, který by nadchl labužníka, ale dostatečně ilustruje několik závažných vlastností zmíněných adresovacích módů. Protože opakování je matkou moudrosti, porovnáme si tyto dva mnemonické kódy:

```
LD HL, A11CH
LD HL, (A11CH)
```

Na pohled se liší pouze závorkami. Avšak pro assembler má tato diference zásadní význam. U první instrukce jde o přenos dvou bajtů A1H a 1CH do párového registru HL - to je přímý dvoubajtový přenos. Ve druhém případě se však kontaktujeme*s pamětí - číslo A11CH v závorce znamená první adresu, z níž bude proveden přenos dat (čili přenos obsahu uvedené adresy). Po provedení této instrukce bude bajt z adresy A11CH přenesen do registru L a bajt z adresy o 1 vyšší (tedy z A11DH) do reg. H (napřed z nižší do nižšího, pak z vyšší do vyššího registru). U instrukcí jednobajtového přenosu dat LD (HL),EEH a LD (HL),FFH je třeba si uvědomit, že obsah reg. HL se vlivem přenosu nemění. V reg. HL zůstává číslo adresy, na niž se datový bajt přenáší, nezměněno. Změní se však obsah adresy určené obsahem reg. HL.

KROK 3: Určete, jaký HD obsah budou nakonec mít registry H a L a co bude na adresách A11CH a A11DH.

KROK 4: Zjistěte si z monitoru správnost svých odpovědí: H = EEH, L=FFH, (A11CH) = FFH, (A11DH) = EEH.

EXPERIMENT č.3

Sestavení programové smyčky - v uvedeném případě jde o časovači smyčku za použití instrukce JP NZ, XXXX.

Rutina A

```
A120  0E00          LD C, 00H       ;Vynulování reg. C
A122  00     SMYC:  DEC C           ;Sníženi obsahu reg. C o 1
A123  C222A1        JP NZ, SMYC     ;Když je reg. C různý od nuly, skok
                                    ;na adresu SMYC
A126  C9            RET             ;Když je reg, C = 0, návrat
```

Rutina B

```
A130  0600          LD B, 00H       ;Vynulováni reg. B
A132  0E00   SMYC1: LD C, 00H       ;Vynulování reg. C
A134  0D     SMYC2: DEC C           ;Obsah reg. C-1
A135  C234A1        JP NZ, SMYC2    ;Když reg. C není 0, skok na SMYC2
A138  05            DEC B           ;Obsah reg. B-1
A139  C232A1        JP NZ, SMYC1    ;Když reg. B není 0, skok na SMYC1
A13C  C9            RET             ;Návrat
```

KROK 1: Zapište do počítače obě rutiny a zkontrolujte správnost zápisu.

KROK 2: Obě rutiny si probereme trochu podrobněji, abyste přesně pochopili jejich funkci. Rutina A je jednoduchou časovači smyčkou. Novinkou pro vás je použití návěští (angl. *label)* v zápisu programu - SMYC, SMYC1, SMYC2. Tato návěští jsou nesmírnou pomůckou při programování v assembleru. Jak jste se v úvodní části dozvěděli, je každý program protkán instrukcemi skoků a volání. Pokud bychom pro adresy těchto instrukcí používali jenom číslo, měli bychom „v tom" za chvíli slušný zmatek. Pojmenujeme-li si je však nějakým symbolickým názvem, struktura programu bude rázem přehlednější, „čtivější". Třeba adresu rutiny, která pohybuje bodem na obrazovce vpravo nazveme PRAHYB, vlevo VLEHYB, rutinu tisku TISK apod.

To však není vše, čím návěští usnadňují programátorům práci. Generátory strojového kódu mají jednu skvělou přednost. Dokáží s návěštími pružně pracovat tak, že mezi návěští a „stejnojmenné" instrukce skoku a volání můžeme libovolně vkládat nebo odtud naopak ubírat bajty dle libosti, aniž se musíme starat o to, zda skok nebo voláni budou provedeny správně. Jak sami vidíte, nemusíme na adrese A123H psát JP NZ, A122H, stačí pouhé JP NZ, SMYC. Mezi adresy A122H a A123H můžeme cokoli vklá-

dat; kdykoli však dojde řada na provedení instrukce JP NZ,SMYC, pak v případě splněné podmínky NZ bude program nasměrován vždy na adresu SMYC. Pokud by generátor tuto schopnost neměl, museli bychom během všech oprav a přesunů instrukcí v rutinách stále měnit i adresy skoků a volání, což by bylo více než nepříjemné a úmorné. Překladač nám také může vypsat všechna dosud použitá návěští i s jejich adresami, což dále zvyšuje orientaci, zvláště v delším programu.

Nyní k samotné rutině A. Reg. C po snížení svého obsahu z nuly bude obsahovat 255 (FFH). Protože podmínka v instrukci JP NZ,SMYC stanoví, že pokud obsah reg. C nebude roven nule, má program skočit vždy na adresu SMYC, učiní tak vždy, dokud se instrukcí DEC C jeho obsah postupně (256krát) nesníží až na nulu. Poté bude program pokračovat na adresu s instrukci RET a vrátí řízeni zpět k adrese volání rutiny. Tak vlastně program vykonal 256 skoků v časovací smyčce. Její trvání můžeme řídit stanovením vstupní hodnoty reg. C. Čím nižší tato hodnota bude, tím kratší bude i doba provedení smyčky.

Pro programy tohoto typu je dobré si nakreslit vývojový diagram. Pokud se vám stane, že máte program zaplaven podmínkami a odskoky, v nichž se už sami přestáváte vyznávat, diagram si nakreslete. Pomůže vám nejen najít chyby, ale někdy dodatečně zjistíte, že program můžete i výhodně zkrátit. Rozličné časovací smyčky se v programech objevuji velmi často. Jsou nezbytné všude tam, kde probíhá nějaký proces, jehož rychlost potřebujeme zbrzdit, resp. nastavit přesný čas provedení nějaké části programu, nebo když čekáme po určitý časový úsek na nějakou datovou informaci (z periférií) apod. Podobné smyčky použijeme také tehdy, Chceme-li, aby nějaké operace proběhly vícekrát po sobě, při inicializaci obsahu registrů nebo míst paměti apod.

KROK 3: Rutina B je příkladem možnosti začlenění dvou smyček do sebe. Smyčka SMYC2 je vnitřní, SMYC1 vnější. SMYC2 se provede vždy 256krát. Délku časové prodlevy způsobené touto časovací rutinou nastavujeme vstupní hodnotou reg. B. Takových smyček bychom mohli prokombinovat mnoho. V dalších částech učebnice se dozvíte, jak sestavovat časovací smyčky i jiným, úspornějším a efektivnějším způsobem.

KROK 4: Pro úplnost si ještě ukážeme, jak do rutiny B začlenit ještě jednu vnější smyčku:

```
A12E 1630              LD D, 30H        ;inicializace proměnné vnější smyč.
; Na adresách A130H-A13BH je umístěna rutina B (bez instrukce RET)
;
A13C 15                DEC D            ;Snížení obsahu reg. D o 1
A13D C230A1            JP NZ,A130H      ;Pokud není D= 0, skok na obě vnitř-
                                        ;ní smyčky
A140 C9                RET              ;Je-li D = 0, návrat
```

KROK 5: Pokud jste novicem programování, měl byste se pokusit zakreslit vývojový diagram celé rutiny se všemi třemi smyčkami. Hodně vám napoví.

KROK 6: Spusťte program od adresy A12EH. Jak sami zjistíte, doba jeho provedení se značně protáhla. Čekáte-li však na jeho ukončení déle než minutu, máte v programu ně-

jakou chybu. Musíte provést RESET počítače a zkusit to znova, teď už s pečlivější kontrolou svého zápisu.

KROK 7: Zkuste měnit bajt na adrese A12FH, abyste si ověřili, jak jeho hodnota ovlivní dobu provedení rutiny.

EXPERIMENT č. 4

Blokový přenos s instrukcí LDDR.

```
A150  2175A1          LD HL,A175H     ;Nejvyšší zdrojová adresa bloku,
                                      ;z něhož budou data odebírána
A153  116FA1          LD DE,A16FH     ;Nejvyšší adresa určeni nového
                                      ;uložení přenášeného bloku dat
A156  010500          LD BC,0005H     ;Do reg. BC počet bajtů přenosu
A159  EDB8            LDDR            ;Blokový přenos
A15B  C9              RET             ;Návrat
```

KROK 1: Vložte program a zkontrolujte jeho zápis.

KROK 2: Podle obsahu párového registru BC jste jistě poznali, že jde o přenos pěti bajtů, který vmžiku proběhne v tomto sledu:

bajt z adresy:	je přenesen na adresu:
A175H	A16FH
A174H	A16EH
A173H	A16DH
A172H	A16CH
A171H	A16BH

KROK 3: Uložte na adresy A171H-A175H postupně tyto bajty: AAH, BBH, CCH, DDH, EEH. Spusťte program a poté si prohlédněte obsah adres, na něž mají být uvedené bajty přeneseny. Měly by se vám na nich objevit ve stejném sledu. Můžete si vyzkoušet i osazení jiného počtu bajtů zdrojového bloku se změnami obsahu reg. BC, případně i změnami reg. HL a DE, abyste se s instrukcí seznámili co nejdůvěrněji.

Instrukce blokového přenosu dat patří mezi instrukce s tzv. mezními stavy. Např. když zahájíte přenos s obsahem reg. BC = 0000, pak se BC převrátí do stavu FFFFH a instrukce provede přenos 65536 bajtů (přičemž zničíte všechny programy, které v paměti máte!). V takovém případě se však nepřenese všech 65536 bajtů, ale jen tolik, kolik jich bude přeneseno do momentu, než instrukce přenosu přepíše samu sebe. Na to je třeba při stavbě programu pamatovat.

EXPERIMENT č. 5

Obsahuje analýzu instrukce LDI. Nově zařadíme další podmíněné instrukce přímého skoku JP Z,XXXX a JP PE, XXXX a logickou instrukci OR r.

```
A180  21A0A1          LD HL,A1A0H     ;Zdrojová adresa
A183  11C0A1          LD DE,A1C0H     ;Adresa určeni přenosu
A186  011000          LD BC,0010H     ; Počet bajtů přenosu
A189  7E       SMYC:  LD A, (HL)      ;Uložení každého zdroj. bajtu do r. A
A18A  B7              OR A            ;Nastavení indikátoru nuly Z v reg. F
                                      ;na log.1, když obsah reg. A = 0
A18B  CA93A1          JP Z, KONEC     ;Je-li A = 0, skok na adr. KONEC
A18E  EDA0            LDI             ;Přenos bajtu
A190  EA89A1          JP PE, SMYC     ;Dokud BC není 0, skok na adr. SMYC
A193  C9       KONEC: RET             ;Návrat
```

KROK 1: Zapište program do počítače a proveďte kontrolu správnosti zápisu.

KROK 2: Nyní se podíváme na instrukce, které ještě neznáme:

OR A - logická operace OR akumulátoru se sebou samým. Např. OR C by byla operace OR akumulátoru s registrem C. Indikátor Z = 1 tehdy, když výsledkem operace je nula - a to nastane jen tehdy, když obsahy obou bajtů na operaci zúčastněných jsou nulové. Této vlastnosti funkce OR se velmi často využívá v testech, kterými zjišťujeme, zda obsahy nějakých dvou bajtů (nebo obsah akumulátoru) jsou nulové. Logickými instrukcemi se budeme zabývat později. Tento test, který patří k programátorským finesám, si zapamatujte pro svou budoucí praxi.

JP Z,XXXX - podmíněná instrukce přímého skoku na adresu XXXX. Provede se tehdy, je-li indikátor Z = 1, jinými slovy tehdy, je-li výsledkem poslední předchozí operace, která ovlivňuje stav indikátoru Z, nula.

JP PE, XXXX - podmíněná instrukce přímého skoku na adresu XXXX. Dokud obsah reg. BC není nulový, indikátor P/V je ve stavu log.1 - podmínka PE je splněna a provede se skok na adresu XXXX. Při BC = 0 platí podmínka PO. Tento test má u instrukcí blokového přenosu a prohledávání zvláštní postaveni (nesouvisí se zjišťováním parity - viz použití podmínky PO v programu 2 experimentu 4 v kapitole 6). Podmínkami PE a PO, vztahujícími se ke zjištění parity, se zabývá hlavně část věnovaná logickým instrukcím.

Vzpomeňte si, že vedle těchto podmíněných skokových instrukcí známe ještě jednu - JP NZ,XXXX. Jak jste si už určitě uvědomili, mnemonika Z80 řadí instrukce, které jsou si něčím podobné, do skupin tak, že nám umožňuje snadnou orientaci v celém instrukčním souboru Z80.

Podmíněné instrukce mají jedno společné - všechny testují stav některého z indikátorů v reg. F a provedou se jedině tehdy, je-li jimi vytčená podmínka splněna. Jinak jsou ignorovány. Ovšem ne tak zcela - jak vysvítá z tabulky, v níž jsou uvedeny časy potřebné pro provedení jednotlivých instrukcí, i v případě nesplnění podmínky určitý čas zabere dekódování instrukce a zjišťování, zda je podmínka splněna. Kromě instrukcí typu JP je tento čas kratší, než v případě podmínky splněné. Proto jsou u podmíněných instrukcí

(a u těch, které mají nějakou „skrytou" podmínku ve své podstatě) uvedeny vidy dvě časové hodnoty (u JP výjimečně jen jedna).

KROK 3: Proveďme si analýzu našeho programového experimentu. První tři instrukce nastavuji obsahy párových registrů pro blokový přenos dat instrukcí LDI. Další dvě instrukce zjišťují, zda obsahem přenášeného bajtu (z adresy určené číslem v reg. HL) není nula. Bajt z této adresy je přenesen do reg. A, aby s ním hned nato byla provedena logická operace OR A. Je-li obsah reg. A = 0, podmínka instrukce JP Z,KONEC je splněna a program přeskočí na adresu KONEC, na níž je instrukce návratu RET- program se • ukončí. Není-li podmínka splněna, provede se vlastní přenos jednoho bajtu instrukcí LDI. Jak už bylo výše uvedeno, indikátor P/V se provedením instrukce dostane do stavu log.1, přičemž je nutno testovat paritu (podmínky PE a PO), nikoli aritmetické přeplněni (M a P). Dokud obsah reg. BC není roven nule, provede se vždy skok na adresu SMYC. Jakmile je reg. BC = 0, indikátor P/V bude rovněž nulový a instrukce JP PE, SMYC se neprovede; program bude pokračovat na instrukci RET a ukonči se.

Test na adrese SMYC je ukázkou toho, jak je možno využít nějaké hodnoty bajtu (zde je to nula) k tomu, aby se řetěz prováděni instrukcí přerušil a program přešel jinam, jak mu navelíme v podmínce. Kdykoli se tedy mezi našimi 16ti přenášenými bajty objeví nulový bajt, program se ukonči. Toho se dá využít např. při sestavování nápisů na obrazovce. Dejme tomu, že jednotlivá slova (jejich písmena budou bajty s obsahem kódů ASCII) oddělíme nulami. Po testu nuly (když Z = 1) se programové řízení převede na rutinu zadáni nové pozice začátku tisku dalšího slova na obrazovce. Z této rutiny se program vrátí na přenos dalších písmen na obrazovku. Jakmile v přenosu znova narazí na nulu, přejde na zadání nové pozice tisku, pak zase zpět atd., dokud nebude určený počet slov vypsán. Tento počet zase určíme nějakým testem, který zjistí, jestli je vypsáno vše, co bylo třeba, a po splnění podmínky se převede řízení programu na jinou programovou rutinu, jak sami stanovíme.

Z toho přibližně vidíte, jak takový program ve strojovém kódu pracuje - jedna rutina za druhou jsou aktivovány tak, aby program vykonával stanovené požadavky. Jde vlastně nejen o detailní vytvoření samotného programu, ale především jeho struktury, která bude funkční, optimální, efektivní a bezchybná. Bez předchozího promyšlení „generálního strategického plánu" stavby našeho programu se v něm snadno a brzy ztratíme. Strukturou programu je zde myšleno blokové schéma programu, sestavené z jednotlivých rutin s uvedením jejich funkcí, vstupních podmínek a výstupních parametrů.

KROK 4: Uložte do paměti od adresy A1A0H libovolné nenulové bajty v počtu 16 a spusťte program. Po jeho skončení si prohlédněme obsahy registrů HL, DE, BC. Měly by obsahovat čísla A1B0H, A1D0H a 0000H. Tak je potvrzena správná funkce programu

KROK 5: Nyní na uvedené adresy uložte tyto bajty:

A1A0H-10H
A1A1H-0FH
A1A2H-0EH
A1A3H-00H
A1A4H až A1AFH-FFH

Spusťte program. Co se stane, až dojde k testu nulového bajtu z adresy A1A3H? Program se samozřejmě zastaví, přičemž obsahy registrů budou tyto: BC = 000DH, HL = A1A3H a DE = A1C3H. Přenos bajtů s obsahem FFH se už neprovede. Tím je potvrzena naše předchozí analýza programu.

Pokud se ptáte, k čemu je nám opakování instrukce LDI nebo LDD, když máme možnost použít LDIR nebo LDDR, které se opakují samy, odpověď je jednoduchá. Do přenosu prováděného instrukcemi s automatickým opakováním nemáme možnost zasáhnout, tedy ani testovat jednotlivé přenášené bajty, kdežto aplikace instrukcí LDI LDD zařazení testů či dalších operací umožňují.

EXPERIMENT č. 6

V něm si ukážeme, kolik jakých instrukcí navíc bychom museli použít pro náhradu instrukce LDIR, kdyby ji mikroprocesor Z80 neměl ve svém instrukčním souboru (např. mikroprocesor 8080 tuto instrukci nezná). Zároveň si porovnáme i čas, po který se provádí instrukce LDIR s časem, který zabere její náhrada.

Rutina A

```
A1D0  2100A2          LD HL, A200H    ;Inicializace registrů k provedení
A1D3  1101A2          LD DE,A201H     ;instrukce LDIR
A1D6  016400          LD BC, 0064H
A1D9  EDB0            LDIR
A1DB  C9              RET
```

Rutina B

```
A1D0  2100A2          LD HL,A200H     ;Obvyklá inicializace registrů
A1D3  1101A2          LD DE,A201H
A1D6  016400          LD BC, 0064H
A1D9  7E        SMYC: LD A, (HL)      ;Obsah adresy HL do reg. A
A1DA  12              LD (DE),A       ;Přenos obsahu reg. A na adresu (DE)
A1DB  23              INC HL          ;Zvýšení HL o 1
A1DC  13              INC DE          ;Zvýšení DE o 1
A1DD  0B              DEC BC          ;Snížení BC o 1
A1DE  78              LD A,B          ;Test obsahu BC; je B i C rovno 0?
A1DF  B1              OR C            ;a je tedy obsah reg. BC = 0?
A1E0  C2D9A1          JP NZ, SMYC     ;Pokud ne, skok na adr. SMYC
A1E3  C9              RET             ;Návrat
```

KROK 1: Obě rutiny přenášejí blok 100 bajtů. Rutina A však zabírá jen 14 bajtů paměti a její provedení trvá 2095 taktů, tj. 2095*0,000004 = 0,00838 vteřiny.

Oproti tomu rutina B zabírá 22 bajtů paměti a její provedení trvá 5000 taktů, tj. 0,02vteřiny, což je více než dvakrát tolik. S rostoucím počtem přenášených bajtů poroste úměrně i doba provedení přenosu bloku dat rutinou B.

KROK 2: Ozkoušejte si funkci obou rutin, abyste zjistili, že jsou skutečně ekvivalentní. Zvyšováním počtu bajtů přenosu v rutině B si můžete ověřit, že doba jejího provedeni začne být brzy patrná.

V rutině B je několik nových instrukcí zvyšování a snižování obsahů párových registrů. Probereme si je později. Nerozlučný pár instrukcí testu obsahu dvou registrů logickou funkcí OR už znáte z předchozích experimentů.

KAPITOLA 2
ADRESOVACÍ MÓDY Z80
2. ČÁST

I když vám minulá kapitola lehce přiblížila i jiné typy instrukcí, než typu LD s jednobajtovým operačním kódem, stále stojíte teprve na začátku. Od této kapitoly přidáme plyn. Kdyby se vám hned zkraje přehřál motor, nechte ho trochu vychladnout a projeďte se ještě jednou kapitolou předchozí - zřejmě vám z ní všechno ještě neslehlo. Jakékoli šizení výukových dávek se dříve či později negativně projeví. A nakonec byste mohli zanevřít i na strojový kód, který by v tom ovšem byl zcela nevinně. Před tím, než si probereme zbývající polovinu adresovacích módů, musíte se seznámit s dvojkovou aritmetickou komplementací, která je základem pro pochopení indexovaného a relativního adresování užívaného několika typy instrukcí. Proto následující řádky budou poněkud hutnější pro ty z vás, kteří ve škole více holdovali přestávkám matematice. Snad vám trochu klidu přinese ujištěni, že mnohem více než o matematiku samu, půjde o logiku uvažování (nakonec - budeme se pohybovat jen v rozmezí čísel -128 až +127).

JEDNIČKOVĚ KOMPLEMENTÁRNÍ BINÁRNÍ REPREZENTACE

Na ní není nic složitého. Jedničkový komplement binárního čísla obdržíme prostým převrácením hodnot (inverzí) všech jeho bitů. Tak např. binárně vyjádřený bajt 00000000 (tedy nula) má jedničkový komplement 11111111 (FFH). Jedničkovou komplementaci můžeme provádět řadou instrukcí Z80.

DVOJKOVĚ KOMPLEMENTÁRNÍ BINÁRNÍ REPREZENTACE

Zatím víme, že číslo, které se „vejde" do jednoho bajtu, není menší než nula a větší 255, což dává dohromady 256 různých čísel. Rovněž víme, že těchto 256 čísel je 256 možných kombinací jedniček a nul na pozicích osmi bitů jednoho bajtu. Řekněme si hned na začátku, že dvojková komplementace (budeme ji dále značit zkratkou DK) nám v určitých případech umožňuje pracovat s jedním bajtem jako číslem v rozsahu -128 až +127 včetně nuly. V podstatě jde jen o jiný pohled na stále stejný počet 256 kombinací

binárního kódu osmi bitů jednoho bajtu. K čemu je tento doplněk dobrý, si ukážeme hned po objasnění, jak to s tou DK vlastně je.

Abychom si to zkraje trochu zjednodušili, budeme počítat jen se čtyřmi bity. Tak si podstatu DK vysvětlíme na příkladu čtyřbitové DK reprezentace:

Dekadické číslo	HD číslo	4-bitová DK reprezentace
7	7	0111
6	6	0110
5	5	0101
4	4	0100
3	3	0011
2	2	0010
1	1	0001
0	0	0000
-1	F	1111
-2	E	1110
-3	D	1101
-4	C	1100
-5	B	1011
-6	A	1010
-7	9	1001
-8	8	1000

Na první pohled je patrné, že čísla kladná a záporná se liší obsahem nejvyššího bitu. Kladná (a nula) mají tento bit ve stavu log.0, zatímco záporná ve stavu log.1. Zapamatujte si jednu mnemotechnickou pomůcku -v duchu si nulu zařaďte mezi nezáporná čísla. Pak vám nebude dělat potíže uvědomit si, že nezáporných čísel je stejný počet jako záporných, ale kladných (větších než 0) je o 1 méně než záporných.

Normální binární reprezentace 4 bitů by zahrnovala čísla v intervalu 0—15. V případě DK to budou čísla v rozsahu -8 až +7 podle formulky:

-2**(n-1) až + (2**(n-1))-1 (kde n je počet bitů)

Při pohledu na tabulku rychle zjistíte, že kladné DK číslo je identické s dekadickým nekomplementárním vyjádřením jeho binární reprezentace. To ale neplatí pro záporná DK čísla (např. -1 versus 15).

Dvojkovým komplementem čísla X je takové číslo, které přičteno k číslu X způsobí, že výsledek binárního součtu bude o jeden řád vyšší, přičemž všechny zbývající číslice součtu budou rovny nule:

0001	(+ 1)	1010	(- 6)	0101	(+ 5)
1111	(- 1)	0110	(+ 6)	1011	(- 5)
10000		10000		10000	

To znamená, že dvojkovým komplementem binárního čísla 0001 je číslo 1111, čísla 1010 číslo 0110 a čísla 0101 číslo 1011. Přitom je třeba si uvědomit, že ve vlastním provádění operací s osmi bity bajtu je při DK součtu výsledkem 00000000, tedy nikoli 100000000. Jde o nám již známé přeplnění bajtu. Podíváme-li se na celou věc z hlediska bajtových operací, můžeme říci, že vzájemně dvojkově komplementární jsou taková čísla, jejichž součet vynuluje obsah bajtu, do něhož se výsledek součtu ukládá (s následným nastavením indikátoru přenosu CY na log.1).

Dále můžeme jednoduše říci, že DK číslo k číslu danému najdeme tak, že k jeho jedničkovému komplementu přičteme číslo 1:

	JK		DK	
DK čísla 1010 je:	0101	+ 0001	=0110	
DK čísla 0000 je:	1111	+ 0001	=0000	
DK čísla 1000 ??:	0111	+ 0001	=1000	(NE!)

V posledním případě jde o DK čísla -8, které ve čtyřbitové reprezentaci nemá svůj DK (číslo +8 v této reprezentaci není obsaženo). V jakémkoli binárním rozsahu platí vždy, že nejvyšší záporné číslo nemá svůj DK!

Jaké bude nejvyšší kladné číslo osmibitového binárního DK kódu? Protože kladné číslo musí začínat nulou, budou na zbývajících sedmi pozicích jedničky: 01111111 (to je dekadicky +127).

Jaké bude nejnižší DK číslo téhož kódu? Napřed převrátíme stavy všech bitů (10000000) a přičteme číslo 1. Výsledek je 10000001 (dekadicky -127). Z toho je patrno, že může existovat ještě jedno menší číslo (10000000), což je DK reprezentace čísla -128. Tím se opět potvrzuje, že nejvyšší záporné číslo (zde -128) nemá svůj vlastni DK.

Osmibitová binární DK reprezentace v sobě tedy zahrnuje čísla v intervalu -128 až + 127, čímž jsme si potvrdili naše tvrzení z úvodu této kapitoly.

Vše výše uvedené vám pomůže kdykoli určit DK reprezentaci jakéhokoli binárního čísla. V případě, že nejvyšší bit bajtu je nulový (pak i SI znaménka S=0), jde o kladné DK číslo, které je ekvivalentní jeho nekomplementární hodnotě. Je-li nejvyšší bit bajtu ve stavu log.1 (S = 1), jde vždy o záporné číslo DK reprezentace.

Nejrychlejší způsob odvození DK binárního čísla spočívá v jeho konstrukci „zezadu". Všechny nuly zprava až po první jedničku necháme beze změny. Hodnoty všech dalších bitů převrátíme. Ověřte si tuto metodu na výše uvedených příkladech.

DK čísla nejsou nikterak samoúčelná a ve výpočetní technice mají svůj velký význam. Abychom se s nimi seznámili opravdu důkladně, probereme si ještě operace zvané

DVOJKOVĚ KOMPLEMENTÁRNÍ SEČÍTÁNÍ (I ODEČÍTÁNÍ)

Součet DK čísel je velmi triviální:

```
0 0 0 0 0 1 1 1 (+7)
0 0 0 0 0 0 1 0 (+2)
---------------
0 0 0 0 1 0 0 1 (+9)
```

```
1 1 1 1 1 1 0 0 (-4)
0 0 0 0 0 0 1 1 (+3)
---------------
1 1 1 1 1 1 1 1 (-1)
1 1 1 1 1 0 0 1 (-7)
1 1 1 1 0 0 1 1 (-13)
---------------
1 1 1 0 1 1 0 0 (-20)
0 1 1 0 0 0 0 0 (+96)
0 1 0 1 0 0 0 0 (+82)
---------------
1 0 1 1 0 00  0 NELZE!
1 0 1 1 1 0 0 1 (-71)
1 0 1 1 1 0 0 0 (-72)
---------------
0 1 1 1 0 0 0 1 NELZE!
```

K prvním třem příkladům netřeba nic dodávat. V posledních dvou došlo k přeplnění (angl. *overflow*- přetečení) rozsahu osmibitové binární DK reprezentace. Toto přeplnění označujeme jako aritmetické. Kladné DK číslo nemůže být větší než 127, záporné menší než -128. K detekování aritmetického přeplnění používáme stavový indikátor P/V. Dojde-li přitom i k přeplnění binární reprezentace bajtu, změní se také stav indikátoru přenosu CY.

Nyní se vrátíme k adresovacím módům Z80 a probereme si zbývajících pět.

ADRESOVÁNÍ RST

Instrukční soubor Z80 obsahuje 8 zvláštních instrukcí, které převádějí řízeni programu na instrukcemi pevně stanovené adresy: RST 00H, RST 08H, RST 10H, RST 18H, RST 20H, RST 28H, RST 30H a RST 38H. Tyto instrukce jsou jednobajtové, zatímco srovnatelná instrukce volání CALL XXXX je tříbajtová. V tom je jejich přednost. Číslo v instrukci RST je číslem adresy, na niž je po jejím provedení programové řízení převedeno. RST XX se používá pro aktivování zásadních, tedy i velice často užívaných rutin operačního systému počítače. Anglicky se tomuto módu říká poněkud složitě *modified page zero addressing,* což česky znamená adresování s modifikovanou stránkou nula. Tím je řečeno, že adresa, na kterou bude převedeno programové řízení, je v rozsahu jedné stránky paměti XX00H - XXFFH, přičemž vyšší bajt adresy je v případě nulté stránky roven nule (XX = 00). Podobně jako u instrukcí volání (viz kapitola 3), je provedení instrukcí typu RST provázeno dvěma přenosy dat - adresa skoku je přenesena do reg. PC (bezprostřední rozšířené adresování) a na spodek zásobníku jsou přeneseny 2 bajty (nepřímé rozšířené adresování), které obsahují adresu návratu (to je první adresa za instrukcí RST); přitom je obsah reg. SP snížen o 2.

IMPLIKOVANÉ ADRESOVÁNÍ

U tohoto adresování se jednotlivé skupiny instrukcí vždy vztahují k jednomu registru mají jej v sobě zahrnut, neboli implikován). Název registru je v zápisu instrukce „zamlčen“. Příkladem takové skupiny instrukcí jsou aritmetické a logické instrukce, v nichž je implikován reg. A. Např. SUB B znamená, že od obsahu reg. A bude odečten h reg. B. Přestože reg. A není přímo obsažen v zápisu instrukce, je v ní implikován, Podobně instrukce XOR H provede logickou operaci XOR mezi registry A a H.

BITOVÉ ADRESOVÁNÍ

Jde o instrukce, které manipulují s jednotlivými bity adres paměti nebo registrů. Bity jsou číslovány od 0 do 7, přičemž nejnižší bit je bit 0. Např. SET 3,B nastaví bit 3 reg. B do stavu log.1. Kromě SET patři do této skupiny RES a BIT.

INDEXOVANÉ ADRESOVÁNÍ

Z80 má dva speciální registry IX a IY, které nelze brát jako registry párové (standardní instrukce neoperují s jejich polovinami), ale jen jako „nerozdělitelné" registry 16-bitové. Adresováni probíhá pomocí uložení čísla odchylky (angl. *displacement)* od adresy v reg. IX, resp. IY do patřičného bajtu instrukce. Tato odchylka (ofset) může být jen v rozmezí -128 až +127 (vida, první příklad užití DK čísel). Např. LD A,(IX+02) - HD kód DD 7E 02 - znamená, že obsah adresy o 2 vyšší, než určuje obsah reg. IX, bude přenesen do reg. A. Instrukce LD (IY + FDH),A (HD kód 7D 77 FD) přenese obsah reg. A na adresu o 3 nižší, než jakou udává obsah reg. IY (FDH je DK reprezentace dekadického čísla -3). Indexované adresování v sobě tedy zahrnuje pravidla adresování nepřímého. Z uvedeného vyplývá, že před použitím takovéto instrukce musíme napřed stanovit adresu IX, resp. IY. Je vhodné ji stanovit tak, aby byla přibližně uprostřed bloku dat, s nimiž budeme často pracovat - v operačních systémech to bývá např. oblast dat systémových proměnných apod.

Pro lepši porozuměni odchylkám v indexovaném adresování si uvedeme pár příkladů (dis je zkratka slova *displacement):*

(IX + dis)	odchylka od adresy(IX) dekadicky
(IX + 7FH)	+ 127
(IX + 09H)	+ 9
(IX + 00H)	= (IX) ,tedy nulová odchylka
(IX + FFH)	- 1
(IX + F0H)	- 16
(IX + C0H)	- 48
(IX + 80H)	- 128

V zápisech assemblerových programů se spíše setkáte s tvarem (IX-2), kde číslo -2 přímo ukazuje hodnotu záporné odchylky. Pro vlastní zápis do paměti v HD tvaru si je však musíte převést na DK číslo FEH. Generátory strojového kódu nám umožňují vkládat čísla ve tvaru binárním, dekadickém i HD. Proto se neděste toho, že byste při vlastním programování museli vše neustále přepočítávat a pro samý přepočet by vám na vlastní tvorbu programu už nezbyl čas. Je však nezbytně nutné pochopit, co, proč a jak se v počítači při provádění jednotlivých instrukcí děje. Bez toho by vám jakýkoli sebelepší generátor nebyl nic platný. Proto nepolevujte!

RELATIVNÍ ADRESOVÁNÍ

Tento mód se vztahuje pouze k jednomu typu instrukci, kterým se říká relativní skoky (angl. *relative jumps).* Jejich mnemonika je JR XX (HD kód 18 XX), kde XX je odchylka zadaná DK číslem (další užití tohoto typu čísel). Instrukce je dvoubajtové - první bajt je operační kód, druhý obsahuje odchylku *(displacement),* opět v rozsahu -128 až + 127. Odečet odchylky má poněkud jiné „startovní" podmínky než v instrukcích indexovaného adresování, kde odečet začíná od adresy určené obsahem reg. IX nebo IY, přičemž adresa (IX) či (IY) sama reprezentuje nulovou odchylku.

Právě tato nulová odchylka relativních skoků leží (pozor, pamatovat!) na adrese, která je ihned za posledním (tedy druhým) bajtem instrukce relativního skoku, což je adresa o 2 vyšší než adresa, na níž instrukce začíná. To je adresa, kterou obsahuje reg. PC po přečtení této dvoubajtové instrukce. Věc si osvětlíme příklady. Stanovme si, že naše instrukce leží třeba na dekadické adrese 1000:

Instrukce	Skok na adresu
JR 09H	1000 + 2+9 = 1011
JR 7FH	1000 + 2+127 = 1129
JR 00H	1000 + 2 + 0 = 1002
JR FEH	1000 + 2-2 = 1000
JR F0H	1000 + 2-15 = 987
JR 80H	1000 + 2-128 = 874

Leží-li tedy instrukce relativního skoku na adrese 1000, můžeme z ni odskočit na jakoukoli z okolních adres v intervalu 874..1129. Povšimněte si jedné zvláštnosti, které se ve svých programech zásadně vyvarujte. Instrukce JR FEH se po každém provedení vrací sama na sebe - program se tak dostane do nekonečné smyčky, z níž není úniku (kromě vnějšího přerušení chodu mikroprocesoru).

Poněkud beze smyslu je zařazení instrukce JR 00 - program pokračuje hned na další adrese za ní. Kdyby tedy instrukce JR 00 v programu nebyla, nic by se nestalo. Je ji však možno uplatnit jako jednu z programátorských fines, kdy programově odjinud měníme obsah bajtu odchylky, což má pak za následek skoky do různých části programu z téhož místa. V tom případě by použití instrukce JR 00 mohlo mít svůj smysl. Uplatnit by se mohla i při ladění časovacích smyček a jiných časově kritických rutin. Ovšem zcela smysl postrádající je JR FFH.

Užití relativních skoků je významné ze dvou hledisek. Oproti přímému skoku JP XXXX, který je tříbajtový, je relativní skok dvoubajtový - jeho užitím šetříme paměť (ale nikoli čas). A nakonec - potřebujeme-li relokovat nějaký program v paměti (celý jej přemístit na jiné adresy), nemusíme odchylky relativních skoků měnit. Zatímco u přímých skoků a řady dalších instrukcí je po relokaci nutno provést adresovou transpozici.

Tak jsme si představili všech 10 adresovacích módů Z80. Příklady jejich programového uplatnění najdete na konci této kapitoly i ve všech experimentech učebnice.

INSTRUKCE PŘENOSU 16-BITOVÝCH DAT

Již jsme se seznámili se všemi skupinami instrukcí 8-bitového přenosu dat. U přenosu 16-bitového jsme poznali instrukce typu LD. K nim dále patří skupina instrukcí, které pracují s přenosem obsahu mezi registry a zásobníkem - PUSH a POP.

Zásobník *(stack)* je oblast paměti, jejíž spodní adresa je adresována obsahem 16-bitového registru SP *(Stack Pointer).* Na adresy tohoto zásobníku můžeme instrukcemi PUSH ukládat a z něj instrukcemi POP odebírat 16-bitová data (2 bajty). Příklad:

Registr SP jsme (instrukcí LD SP, 65100) nastavili na adresu 65100. Dále si ukážeme, co se bude dít v zásobníku při použití instrukcí PUSH a POP:

SP před instrukcí	Instrukce	SP po instrukci	Obsah zásobníku	Obsah spodku zásobníku
65100	PUSH AF	65098	0000000FA000	F
65098	PUSH BC	65096	00000CBFA000	C
65096	PUSH DE	65094	000EDCBFA000	E
65094	PUSH HL	65092	0LHEDCBFA000	L
65092	POP HL	65094	0LHEDCBFA000	E
65094	POP DE	65096	0LHEDCBFA000	C
65096	POP BC	65098	0LHEDCBFA000	F
65098	POP AF	65100	0LHEDCBFA000	0

Instrukcí PUSH a POP se mohou účastnit reg. AF, BC, DE, HL, IX, IY. Z příkladu vyplývá několik zásadních pravidel:

1 - Do zásobníku ukládané 2 bajty se zapisují v pořadí: napřed vyšší a pak pod něj nižší bajt 16-bitového registru.
2 - Po instrukci PUSH se obsah registru SP i spodní adresa zásobníku vždy sníží o 2.
3 - Po každém odběru (POP) se obsah registru SP i spodní adresa zásobníku vždy zvýší o 2.
4 - Ze zásobníku odebírané bajty se přenášejí podle pravidla: co šlo první dovnitř, jde poslední ven. Instrukcí POP se bajtem ze spodku zásobníku napřed naplní nižší, dalším pak vyšší polovina účastnících se registrů.

Příklad je sestaven tak, aby byl co nejnázornější - proto jsou v zásobníku jednotlivá písmena registrů, avšak jen ve funkci symbolů. Ve skutečnosti je to tak, že jakmile jednou uložíme obsah registru do zásobníku, přestává mezi uloženými bajty a registry AF, BC, DE, HL, IX či IY existovat jakákoli spřízněnost. Díky tomu můžeme pomocí těchto instrukcí vzájemně měnit obsahy uvedených registrů. Tak např. chceme-li vyměnit obsahy registrů mezi DE a BC (pro takovou výměnu Z80 žádnou instrukci nemá), zařadíme za sebe instrukce takto:

```
PUSH BC          Profesionálněji ale:    PUSH BC
PUSH DE                                  LD B,D
POP BC                                   LD C, E
POP DE                                   POP DE
```

Druhý sled instrukcí je stejně dlouhý, ale doba jeho provedení je kratší. Na výstupu bude původní obsah reg. BC v DE a původní obsah reg. DE přejde do BC. Zkušení programátoři provádějí se zásobníkem mnohá kouzla - např. jej v některých průbězích programu používají jako oblasti proměnných, změnami obsahu reg. SP vytvářejí několik různých zásobníků pro různá využití jejich obsahu nebo pro různé funkce části programů atd.

Zásobník je podřízen jedné automatické funkci Z80, která je na jednu stranu nezbytná, na druhou stranu je zdrojem mnoha programátorských strastí plynoucích z nepozornosti. Nad tím, co všechno se v průběhu programu v zásobníku odehrává, je někdy těžko udržet přehled. Kdykoli je totiž volána nějaká rutina instrukcí CALL XXXX, je do zásobníku uložena tzv. adresa návratu, na niž se program vrátí poté, co v programu narazí na odpovídající instrukci RET. Obě instrukce si probereme později; v Basicu mají svůj ekvivalent v příkazu GO SUB n a RETURN. Pokud v Basicu použijete RETURN bez toho, že by mu předcházelo GO SUB n, program se zastaví a chybové hlášení vás na tento nedostatek upozorní. Ale když porušíte toto pravidlo v assembleru, instrukce RET si prostě odebere to, co na spodním konci zásobníku právě je, a program skočí na adresu určenou obsahem obou odebraných bajtů. Obvykle následuje krach programu.

Se zásobníkem je nutno pracovat velice obezřetně, protože je jednou z nejčastějších příčin programových kolapsů pramenících z našich chyb. Nakonec je třeba si ještě uvědomit, že obsah adres zásobníku přepisujeme instrukcemi PUSH. Instrukce POP obsah adres zásobníku nemění!

Počítače pracují s vnitřními i vnějšími přerušeními. S mnohými z těchto operací je spojeno dočasné uložení obsahu všech registrů mikroprocesoru do zásobníku. Po skončení operace jsou registry opět nastaveny na své původní hodnoty jejich odběrem ze zásobníku. Tak se obsah paměti pod spodkem zásobníku velmi čile mění. To je třeba mít na paměti především při práci s větším počtem oblastí dat, s nimiž v průběhu programu operujeme jako s paměťovými oblastmi zásobníku. Nepřejeme-li si, aby taková data podlehla destrukci, musí být při práci s nimi přerušení zablokováno (viz kapitola o interfacingu).

Instrukce výměn obsahu registrů

Vyjmenujme si je všechny hned na začátku: EX AF, AF'; EX DE, HL; EX (SP), HL; EX (SP), IX; EX (SP), IY a EXX.

EX je zkratkou anglického slova *exchange* (výměna). Funkci instrukcí si vysvětlíme na instrukci EX DE, HL:

Obsahy registrů před provedením EX DE,HL		Obsahy registrů po provedení EX DE,HL
00	D	02
01	E	03
02	H	00
03	L	01

Instrukce s adresou (SP) danou obsahem registru SP provádějí výměnu mezi obsahem dvou spodních adres zásobníku a párovým registrem v instrukci užitým. Např. při EX (SP),HL se vymění obsah adresy (SP) s obsahem reg. L a adresy (SP+1) s reg. H. Ale obsah samotného reg. SP se nezmění! Mění se jen bajty jím adresované.

Instrukce EX AF, AF' provádí výměnu mezi uvedenými registry obou registrových bank Z80.

Instrukce EXX provádí výměnu mezi registry BC, DE, HL z banky 0 a stejnojmennými registry BC', DE', HL' banky 1 mikroprocesoru.

Oba poslední typy instrukcí výměn jsou zároveň jediné, které provádějí operace s registry druhé registrové banky Z80. Zde jedno velmi důležité upozornění - každý počítač okupuje pro operace probíhající v jeho systému některé z registrů! U každého typu to bývá jiné. Použití těchto registrů se musíte ve svých programech vyhnout, protože by počítač začal „zlobit", nebo by zcela zkolaboval. Někdy lze některé z těchto „zakázaných" registrů přechodně použít za určitých podmínek - to však vyžaduje perfektní znalost systému počítače. I monitory a generátory strojového kódu mívají drobná omezení, která se dozvíte z jejich manuálu. V každém případě se snažte všechna tato omezeni zjistit - jinak byste mohli zcela zbytečně bádat na tím, proč váš program nefunguje, přestože po stránce teoretické je naprosto v pořádku.

CVIČENÍ

Proveďte si je, protože vás včas upozorní na to, čemu jste v textu neporozuměli nebo tomu nevěnovali patřičnou pozornost. Správné odpovědi najdete hned za posledním cvičením.

1. Určete 8-bitový dvojkový komplement následujících 8-bitových binárních čísel:

   a. 00000001
   b. 11011010
   c. 01010101
   d. 11101110
   e. 00001110
   f. 10000000
   g. 11111111

2. a. Jaké je nejvyšší kladné a nejvyšší záporné (v absolutní hodnotě) dekadické číslo v reprezentaci 8-bitového DK?
   b. Jako v bodě a., ale pro 16-bitovou reprezentaci?

3. Určete dekadická čísla reprezentovaná následujícími 8-bitovými DK čísly:

a. 01111000	e. 11110011
b. 10100011	f. 01010100
c. 00000011	g. 11011001
d. 11111111	

4. Určete 8-bitovou DK reprezentaci těchto dekadických čísel:

a.	1	e.	128
b.	16	f.	121
c.	-16	g.	-90
d.	-128		

5. Následující skokové instrukce JP XXXX nahraďte instrukcemi relativního skoku JR XX (čísla jsou dekadická). Odchylku zapište vždy dekadicky i hexadekadicky.

	adresa uložení	instrukce
a.	10000	JP 10020
b.	11111	JP11240
c.	30	JP0
d.	30	JP 39
e.	65500	JP 65374

6. Pro uvedené assemblerové instrukce vyhledejte v tabulkách jejich HD kód (kde je třeba dopsat čísla, dopište):

a. LD A, B	g. LD SP,HL
b. JR 127	h. LD BC, 0109H
c. LD A, (IX+ 06)	i. LD (1030H),BC
d. LD (IX + 06),A	j. LD IX, (1000H)
e. LD (1234H),A	k. PUSH BC
f. LD (IX + 09),33	l. POP IX

7. Zjistěte z tabulek, zda jsou následující instrukce obsaženy v instrukčním souboru Z80:

a. LD AF,BC	f. LD (1234H),56H
b. LD B, (BC)	g. LD (1234H),B
c LD (BC),B	h. LD (DE),45H
d LD IX, IY	i. PUSH 1234H
e. LD HL, BC	j. POP SP

Odpovědi

1.
a. 11111111
b. 00100110
c. 10101011
d. 00010010
e. 11110010
f. neexistuje
g. 00000001

2.
a. 01111111=127; 10000000 = -128
b. 0111111111111111 = (2**15-1) =32767
1000000000000000 = (-2**15) =-32768

3.
a. 120
b. -93
c. 3
d. -1
e. -13
f. 84
g. -39

4.
a. 00000001
b. 00010000
c. 11110000
d. 10000000
e. neexistuje
f. 01111001
g. 10100110

5.
a. JR 18 — JR 12H
b. JR 127 — JR 7FH
c. JR-32 — JR E0H
d. JR 7 — JR 07H
e. JR -12B — JR 7FH

6.
a. 78
b. 18FB
c. DO7E06
d. DD7706
e. 323412
f. DD360933
g. F9
h. 010901
i. ED433010
j. DD2A0010
k. C5
l. DDE1

7. Odpověď je jednoduchá - žádná z těchto instrukcí není mikroprocesorem Z80 proveditelná. Pamatujte si, že soubor instrukcí má své meze, a nelze tedy užívat ekvivalenty jednotlivých instrukcí ve všech jejich možných kombinacích. Uvedené instrukce lze provést jen jejich rozložením na více platných instrukcí.

EXPERIMENT č. 1

Procvičení indexovaného adresování. V komentářích budeme dále používat schematické vysvětlení tvorby programu bez podrobného popisováni vlastní funkce jednotlivých instrukcí.

```
A100  010300          LD BC, 0003       ;3 bajty pro každou řádku
A103  FD2120A1        LD IX, A120H      ;1. adresa tabulky
A107  FD7E00  SMYC:   LD A, (IY + 0)    ;Sloupec 1 do reg. A
A10A  B7              OR A              ;Je reg. A = 0?
A10B  280A            JR Z, KONEC       ;ANO, pak konec
A10D  FD8601          ADD A, (IY + 01)  ;NE, přičtení sloupce 3
A110  FD7702          LD (IY + 02), A   ;Součet do sloupce 3
A113  FD09            ADD IY, BC        ;IY je další řádka tabulky
A115  18F0            JR SMYC           ;Skok na adr. SMYC; opakování
A117  C9      KONEC:  RET               ;Návrat
```

KROK 1: Zkontrolujte správnost svého zápisu do počítače.

KROK 2: Tento program sečítá čísla umístěná ve dvou sloupcích jedné řádky a výsledek každého součtu uloží do sloupce 3 téže řádky. Tabulka je v paměti uložena od adresy A120H.

	adresa	sloupec 1	sloupec 2	sloupec 3
řádka 1	A120H	01	02	?
řádka 2	A123H	10	04	?
řádka 3	A126H	23	13	?
řádka 4	A12CH	00 (bajt	00 zastaví program	)

Jde tedy v podstatě o programovou simulaci stavby takovéto tabulky. Její rozsah závisí na umístění bajtu 00 v paměti. Čísla ze sloupců 1 a 2 musíme samozřejmě uložit do paměti předem na adresy uvedené ve sloupci adres a adresy vždy o 1 vyšší, než je adresa ve sloupci adres uvedená. Výsledek prvního součtu bude na adrese A122H, druhého na adrese o 3 vyšší atd.

KROK 3: Uložte čísla tabulky do paměti a vyzkoušejte funkci programu prohlédnutím obsahu adres s výsledky součtů. Z uvedené aplikace registru IY je patrná velmi pohodlná manipulace s daty dvourozměrné tabulky. Řádka tabulky je specifikována obsahem reg. IY, zatímco s daty se pracuje pomocí jeho odchylky.

EXPERIMENT č. 2

V tomto experimentu si ukážeme, že při sestavováni programu, který má plnit zadaný úkol, můžeme postupovat různými způsoby. Jinými slovy - k jednomu cíli vede větší počet cest. Jednou z velmi efektních programovacích technik je *self-modification.* Při ní dovedně řadíme instrukce tak, že po spuštění programu jsou některé z nich modifikovány (přepisovány) jinými instrukcemi. Tato technika je velmi náročná a nikdo vám nemůže zaručit, že ji ovládnete. Na druhou stranu se však lze obejít i bez ní. V profesionálních programech se příliš často nepoužívá. Zkušení programátoři se k ní uchylují, jen když není zbytí. Hlavním záporem této techniky jsou potíže, které nastanou při změně aplikačních podmínek. Rozbor takového programu je nesmírně obtížný, ne-li nemožný, což stoupenci této techniky považují za výhodu, protože tak docílí jednoho z nejvyšších stupňů utajení programu. Dlužno podotknout, že leckdy jej tak utají i sami před sebou. Techniku samu *lze* použít pochopitelně jen v pamětech typu RAM. V pamětech ROM se s ni tedy setkat nemůžete. Výhodou techniky je především nižší spotřeba paměti.

Rutina A1

```
A2FD  010700          LD BC, 0007     ;Počet sloupců na řádce
A300  1E06            LD E, 06        ;Čítač počtu řádek
A302  F02180A3        LD IY, A380H    ;Adresa 1. bajtu řádky
A306  2111A3          LD HL, A311H    ;Adresa bajtu odchylky
                                      ;v instrukci ADD A,(IY + d)
A309  3600      řádka: LD (HL),00     ;Inicial. odchylky na adr. (HL)
A30B  3E00            LD A, 00        ;Inicial. reg. A
A30D  1606            LD D, 06        ;Počet sčítanců na řádce
A30F  FD86d     sloup: ADD A, (IY + d) ;Přičtení; d se mění programem
A312  34              INC (HL)        ;Zvýšení odchylky d na adr. (HL)
A313  15              DEC D           ;Snížení čítače řádky o 1
A314  20F9            JR NZ, sloup    ;Když D není 0, další součet
A316  FD7706          LD (IY + 06),A  ,Když D = 0, součet do sloupce 7
A310  FD09            ADD IY,BC       ;IY na 1. bajt další řádky
A31B  1D              DEC E           ;Snížení čítače řádek o 1
A31C  20EB            JR NZ, řádka    ;Když E není 0, na další řádku
A31E  C9              RET             ;Když E = 0, návrat
```

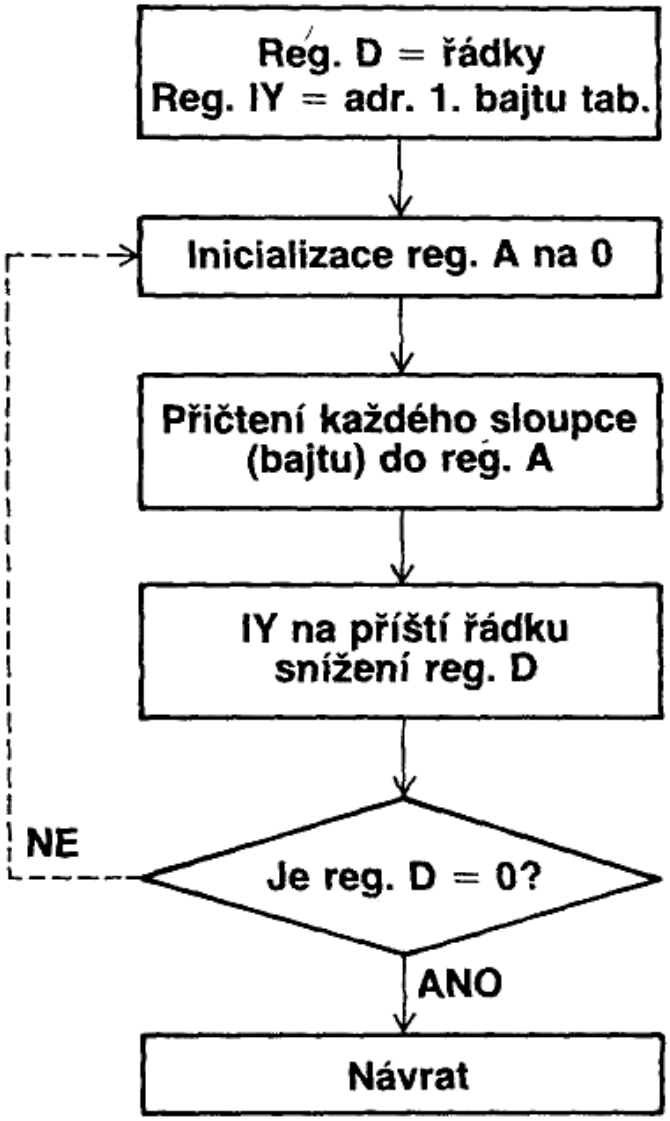
Rutina A2

```
A320  010700          LD BC, 0007
A323  1E06            LD E, 06
A325  FD2180A3        LD IY, A380H
A329  3E00    řádka:  LD A, 00
A32B  FD8600          ADD A, (IY+00)
A32E  FD8601          ADD A, (IY + 01)
A331  FD8602          ADD A, (IY + 02)
A334  FD8603          ADD A, (IY + 03)
A337  FD8604          ADD A, (IY + 04)
A33A  FD8605          ADD A,  (IY + 05)
A33D  FD7706          ADD (IY + 06),  A
A340  FD09            ADD IY, BC
A342  1D              DEC E
A343  20E4            JR NZ, řádka
A345  C9              RET
```

Rutina A3

```
A360  1E06            LD E, 06
A362  FD2180A3        LD IY, A380H
A366  3E00    řádka:  LD A, 00
A368  1606            LD D, 06
A36A  FD8600  sloup:  ADD A, (IY+00)
A36D  FD23            INC IY
A36F  15              DEC D
A370  20F8            JR NZ, sloup
A372  1D              DEC E
A373  FD7700          LD (IY+00), A
A376  FD23            INC IY
A378  10EC            JR NZ, řádka
A37A  C9              RET
```

KROK 1: Při prvním pohledu na tyto tři programy zjistíte, že co do výsledku své činnosti jsou naprosto shodné. Do paměti je umístěna tabulka šesti řádek v šesti sloupcích od adresy A380H. Podobně jako v experimentu předešlém, i tento sečítá čísla na jednotlivých řádkách a výsledek umisťuje vždy do jejich posledního sloupce. Programy se liší rozsahem paměti, kterou zabírají, i časem provedení. Pro všechny tři programy platí tento vývojový diagram:



RUTINA A1 - Hlavním algoritmem rutiny je modifikace bajtu odchylky dis přímo v instrukci ADD A,(IY+dis). Reg. HL je naplňován obsahem adresy, na níž leží třetí bajt instrukce ADD, tedy odchylkou. Nejdříve je odchylka inicializována na nulu, vynulován je i akumulátor. V průběhu čítání sloupců je obsah adresy (IY + dis) přičítán k akumulátoru a odchylka vždy zvýšena o 1 instrukcí INC (HL). Po sečtení čísel ve sloupcích řádky se čítač sloupců vynuluje. To je detekováno instrukcí JR NZ, která způsobí převedení řízení programu na výpočet další řádky.

Program modifikuje sám sebe. Týká se to těchto čtyř instrukcí: LD HL,A311H; LD (HL),00; ADD A,(IY + dis); INC (HL).

Dvě z těchto instrukcí mění program, který vlastně zvažuje sám sebe jako svá data. Znovu opakuji, že užití této techniky si mohou dovolit jen absolventi „vyšší dívčí" - ale i ti mají co dělat, aby tvorbu modifikujícího se programu přežili ve zdraví.

Při zápisu programu v editoru assembleru využívejte adresovou symboliku v míře co největší. V této rutině např. místo LD HL,A311H můžete použít ekvivalentní tvar LD HL,sloup + 2.

RUTINA A2 - Zde jsou odchylky řazeny postupně za sebou v jednotlivých instrukcích ADD A,(IY + dis). Jak vidno, tento postup zabere víc paměti než předchozí, ale na druhou stranu je velmi přehledný. Pokud bychom pracovali s rozlehlejší tabulkou, stal by se však tento způsob řazení instrukcí neudržitelný.

RUTINA A3 - Zde je dvourozměrná tabulka převedena do jednorozměrného pole, jehož indexem je registr IY. Data jsou tedy pojímána sekvenčně. Podmínkou užití tohoto způsobu načítání dat je, že data musejí být v paměti řazena za sebou. Pokud by data byla v paměti všelijak rozházená, tento postup by nebyl k ničemu.

Zařadíme si nový termín z oblasti programování - flexibilita. Čím je stupeň obtížnosti modifikace programu nižší, tím je program flexibilnější (modifikovatelnější a také „čitelnější"). Sebe modifikující programy mají flexibilitu nejnižší, často nulovou.

Lze říci, že čas a prostor jsou u programů nepřímo úměrné. Čím více prostoru paměti spotřebujeme, tím je program rychlejší a naopak (ale nemusí to platit vždy!).

KROK 2: Uložte do počítače rutinu A1 a krokujte jí pomocí vašeho monitoru. Největší pozornost věnujte adrese A311H, na níž je uložena odchylka v instrukci ADD A,(IY + dis). Pro zpracování dat tabulky vložte do paměti tato data:

Adresa	Sl. 1	Sl. 2	Sl. 3	Sl. 4	Sl. 5	Sl. 6	Sl. 7	
A380	01	02	03	04	05	06	X	řádka 1
A387	02	02	02	02	02	02	X	řádka 2
A38E	01	03	01	03	01	03	X	řádka 3
A395	03	03	03	03	03	03	X	řádka 4
A39C	08	08	01	01	08	08	X	řádka 5
A3A3	04	04	04	04	08	08	X	řádka 6

Písmeno X znamená místo uložení výsledku součtu.

KROK 3: V tomto experimentu jste se poprvé setkali s tím, že programování je stejně věda jako umění. Způsobů, jak napsat nějaký program pro daný účel, je vždy víc. Který z možných zvolíme, záleží na mnoha okolnostech. Abyste při tvorbě programu nepropadli pocitům méněcennosti, mohu vás ujistit, že i ti nejlepší programátoři přepisují své assemblerové rutiny dvakrát, třikrát, ale i desetkrát. Programování je opravdu tvůrčím procesem, k jehož hlavním znakům patří, že nic v něm neplatí absolutně.

EXPERIMENT č. 3

V něm si ukážeme užití instrukcí PUSH, POP a instrukcí výměn.

```
A130    EX AF.AF'     ;Výměna obsahu reg. AF a AF'
        EXX           ;Výměna obsahu párových reg.
        PUSH AF       ;Obsah reg. AF do zásobníku
        PUSH BC       ;Obsah reg. BC do zásobníku
        PUSH DE       ;Obsah reg. DE do zásobníku
```

```
PUSH HL         ;Obsah reg. HL do zásobníku
EX AF, AF'      ;Výměna obsahu reg. AF a AF'
EXX             ;Výměna obsahu párových reg.
POP HL          ;Obsah zásobníku do reg. HL'
POP DE          ;Obsah zásobníku do reg. DE'
POP BC          ;Obsah zásobníku do reg. BC
POP AF          ;Obsah zásobníku do reg. AF'
RET             ;Návrat
```

KROK 1: Pro zápis tohoto jednoduchého programu užijte monitor, s jehož pomocí budete do paměti zapisovat HD kódy jednotlivých instrukcí. Tyto kódy jsem do programu záměrně nezařadil, abyste se naučili používat převodních tabulek.

KROK 2: Zapište program a zkontrolujte správnost jeho zápisu.

KROK 3: Nyní je na vás, abyste zvolili adresu zásobníku, čili obsah registru SP. Pro tento účel jsou v instrukčním souboru Z80 instrukce: LD SP,HL; LD SP,IX; LD SP,IY; LD SP, NNNN; LD SP, (XXXX). Kteroukoli z nich umístěte na začátek programu. Pokud si přesně nevzpomínáte na funkci zásobníku, vraťte se o pár stránek zpátky.

KROK 4: Nyní nastavte obsahy všech zúčastněných registrů. Zkuste třeba tyto hodnoty: A=01; F=02; B=03; C=04; D=05; E=06; H=07; L=08. Obsahy těchto registrů a reg. SP můžete nastavit přímo monitorem.

KROK 5: Krokujte programem a pečlivě sledujte, co se bude dít v registru SP a s obsahem zásobníku.

KROK 6: Zkuste změnit obsahy registrů, abyste se ujistili o tom, že funkce přenosu mezi registry a zásobníkem funguje, jak náleží.

KROK 7: Během krokováni jste měli možnost si povšimnout, co se stane po provedení dvou instrukcí výměn. Z příkladu vyplývá, že když registry banky 1 převedeme do banky 0, můžeme je nastavit na požadované hodnoty (např. použitými instrukcemi typu POP) a tyto registry opět uschovat do banky 1. Opět upozorňuji, že je třeba se vyvarovat toho, abyste při těchto výměnách nepoužili, resp. nezměnili obsahy registrů používaných vlastním systémem počítače. Registry banky 1 můžeme někdy pojmout jako určitý „zásobník", v němž si uschováme číselné parametry pro jejich případné pozdější užití.

KROK 8: Protože Z80 má omezený počet registrů a my v programu neustále pracujeme s hromadou čísel, slouží nám instrukce PUSH a POP pro přechodné uschováni číselných parametrů, jejichž hodnoty budeme ještě potřebovat.

Ještě poznámka k zásobníku - pokud ve svém programu měníte jeho adresu, nezapomeňte před výstupem z programu vrátit registru SP jeho původní hodnotu. Na jím adresovaných bajtech leží adresa návratu z vašeho programu do operačního systému počítače nebo programu, z něhož jste své rutiny volali.

KAPITOLA 3
SKOKY, VOLÁNÍ A NÁVRATY

Všechny tyto instrukce převádějí řízení programu na jiné adresy. Lidově řečeno po provedení těchto instrukcí program „odskočí jinam".

PROGRAMOVÉ ŘÍZENÍ

Program je v podstatě soustavou smysluplně řazených bajtů umístěných na adresách paměti. Těmito bajty mohou být buď instrukce nebo data. Instrukcemi řídíme běh programu, zatímco data nám slouží jako číselné parametry, resp. operandy instrukcí.

Zde je třeba si uvědomit, že počítač neví, které bajty jsou instrukcemi a které daty. Dáme-li počítači příkaz, aby začal provádět instrukce od určité adresy, počítač automaticky dekóduje jednotlivé po sobě jdoucí bajty a snaží se je číst jako instrukce. Pokud by se nám programové řízení „zaběhlo" do oblasti, v níž jsou uložena data, počítač by je bez nejmenšího uzardění „luštil" jako instrukce - pochopitelně by se pak začaly dít věci netušené. Z uvedeného vyplývá jeden problém, s nímž se setkáme vždy, kdykoli budeme chtít „rozluštit" nějaký cizí program. Stejně jako počítači, i nám by se mohlo stát, že bychom oblast dat mylně četli jako instrukce. Zmíněný problém je jedním z největších kamenů úrazu vždy, kdykoli jsme postaveni před nutnost relokace nějakého programu, jehož zdrojový text nemáme k dispozici. Pak nezbývá než program analyzovat opravdu detailně, abychom mohli přesně určit, co jsou data a co instrukce.

K pochopení programového řízení je dále nutno mít na paměti, že instrukce se provádějí postupně směrem od nižších k vyšším adresám. Tato posloupnost je přerušena pouze instrukcemi skoků, volání a návratů, které převedou řízení programu na jimi určenou adresu. Od této adresy je program prováděn opět posloupné, dokud znova nenarazí na jeden ze tři typů uvedených instrukcí (nebo na přerušení, které je však ještě daleko před námi).

Na programovém řízení se podílí registr PC- *Program Counter* (programový čítač), který v průběhu programu obsahuje vždy tu adresu, na niž bude programové řízení převedeno. To znamená, že po dekódování každé instrukce se obsah registru PC zvýši o tolik, kolik bajtů právě dekódovaná instrukce obsahovala.

V případě, že program narazí na instrukci skoku JP XXXX, obsah registru PC se těsně před provedením skoku naplní adresou XXXX. Při relativním skoku JR XX se připočte odchylka XX k obsahu reg. PC. Ten je v tom momentě již o 2 vyšší než adresa

uloženi prvního bajtu dvoubajtové instrukce JR XX. Výsledek výpočtu se převede do registru PC a pak už následuje vlastní skok.

V případě, že program narazí na instrukci volání (CALL XXXX), provede se stejná operace jako v případě skoku, ovšem s tím rozdílem, že do zásobníku se uloží adresa návratu, která je o 1 vyšší než adresa posledního bajtu instrukce volání. Do zásobníku se tedy přenese obsah reg. PC těsně před tím, než je naplněn adresou XXXX. Poté co program narazí na instrukci návratu (RET), převede se automaticky adresa návratu ze zásobníku do registru PC. Tak přejde programové řízení na instrukci, která je umístěna hned za instrukcí volání.

Zde ovšem platí pravidlo, které musíte nutně dodržet- na spodních dvou adresách zásobníku musí být patřičná adresa návratu vždy přesně v tom momentu, kdy program narazí na instrukci RET. Pokud by tomu tak nebylo, program by byl převeden někam jinam (na adresu, která by byla stanovena obsahem dvou bajtů právě se vyskytujících na spodních adresách zásobníku). Tak můžeme říci, že CALL i RET jsou vlastně zvláštními typy skokových instrukcí.

Instrukční soubor Z80 neobsahuje žádnou instrukci, jejíž součástí by registr PC byl. Obsah registru PC však lze ovlivnit třeba právě instrukcí RET s předem nastavenými obsahy dvou spodních bajtů zásobníku - ty můžeme stanovit přímo instrukcemi, jež v sobě obsahují registr SP. Další z možných alternativ je přenos obsahu reg. HL do reg. PC provedením instrukce JP (HL). Ovšem pozor - v obou případech se adekvátně změní i programové řízení!

NEPODMÍNĚNÉ SKOKOVÉ INSTRUKCE

Mnemoniku těchto instrukcí už známe. První z nich je JP XXXX, kde XXXX je adresa, na niž bude převedeno programové řízení (adresou XXXX se naplní registr PC). Tato tříbajtová instrukce neprovádí nic jiného, než že svým 2. a 3. bajtem naplňuje reg. PC. Druhá JR XX má tentýž efekt. Liší se jen tím, že je dvoubajtové a XX neobsahuje absolutní adresu skoku, ale odchylku *(displacement)* ve dvojkově komplementární reprezentaci - viz podrobné vysvětlení v předchozím textu.

K instrukcím typu JP patří ještě instrukce s registrovým nepřímým adresováním: JP (HL), JP (IX), JP (IY). Adresa skoku je zde dána obsahem zúčastněného registru.

Malá poznámka ke skokům a jejich začleňování do programů. Programy tvořte s co nejmenším počtem skokových instrukci. Program protkaný skokovými instrukcemi je značně nepřehledný, zamotává se. Když už je musíte použít, sestavujte strukturu programu tak, abyste nemuseli používat skoky JP, ale JR. Nejenže tím šetříte paměť, ale při relokaci programu si nemusíte dělat starosti s přepisováním absolutních adres instrukcí JP. Výjimku tvoří užití instrukcí JP u časově kritických operací a při testech parity a polarity (stavové indikátory P/V a S) - instrukce JR tyto testy provádět nemohou.

Výstavbu struktury programu můžeme přirovnat k architektonické tvorbě, a to i v takovém detailu, jakým je řešení inženýrské sítě. Z plynového kohoutku nám nesmí téci voda, z vodovodního syčet plyn. Programovou strukturu tvoří jednotlivé podprogramy (subrutiny) s předem stanovenými funkcemi. Jejich hlavní bloky jsou volány instrukcemi CALL XXXX (s následnými instrukcemi návratů). Tak se aktivují základní funkční vazby

programu. Nepromyšlená struktura vás přinutí používat dlouhé přeskoky programem a nakonec vám nezbyde, než do něj zařazovat stále rostoucí (a strukturu zatemňující) počet instrukcí JP.

Mnohé dobré systémové a užitkové programy z ne tak dávné doby byly relokovatelné. Relokovatelnost byla imperativem. Dnes se od ní upouští. S relokovatelností programu musel jeho tvůrce počítat předem a svou tvorbu tomuto požadavku podřídit. Tak mu program „nabobtnal" a ztrácel čistotu své struktury. Ve výsledku mohl být o poznání těžkopádnější, v každém případě byl i delší než programy nerelokovatelné. Máme-li však dobře napsaný zdrojový text assemblerového programu s dobrou strukturou, můžeme jej umístit relativně kamkoli a pomocí referencí k němu dále připojovat pomocné rutiny z tzv. knihovny podprogramů.

PODMÍNĚNÉ SKOKY A STAVOVÉ INDIKÁTORY

Letmo jste se s některými skoky i indikátory seznámili již v první části. Teď si je probereme podrobně.

Existují tři typy instrukcí, které testují hodnotu jednotlivých stavových indikátorů - podmíněné skoky, volání a návraty.

Z80 má v reg. F celkem 6 SI. Instrukcemi můžeme testovat 4 z nich. Dva zbylé slouží interním testům mikroprocesoru; jejich stavy však můžeme sledovat pomocí monitoru. Ve skokové instrukci uvedená podmínka testuje, zda je ten který testovaný SI ve stavu log.0 nebo log.1.

Mikroprocesor Z80 obsahuje tyto stavové indikátory:

Carry flag (indikátor přenosu) - CY

Je ovlivňován především instrukcemi provádějícími součet, odečet, bitovou rotaci a posuv. Tyto operace provádí celá řada instrukcí. V psaném textu se tento SI značí CY, aby se nepletl s reg. C.

Při součtu je CY ve stavu log.1 vždy, když dojde k přeplnění akumulátoru, do něhož se ukládá výsledek součtu. Přeplnění znamená, že akumulátor by po něm měl mít obsah minimálně 100000000, což nejde, protože to je celkem 9 bitů a registr jich má jen 8. Onen 9. bit obrazně řečeno „přepadne horem a zmizí". V součtech vícebajtových však můžeme tento bit přenést do vyššího bajtu a přičíst jej k jeho nejnižšímu bitu. Odtud název indikátor přenosu. Uvedené analogicky platí pro přičítání k reg. HL, IX a IY s tím rozdílem, že jde o operaci 16-bitovou. CY signalizuje „přepad" bitu 15.

Při odečtu se vše odehrává v opačném směru. CY = 1 i tehdy, když výsledek odečtu je (by měl být) menší než 0. Toho se opět využívá ve vícebajtových odečtech, kdy „dolem přepadlý" bit je odečten od obsahu nižšího bajtu.

Jak víme, přičteme—li k registru obsahujícímu FFH číslo 1, jeho obsah se nastaví na nulu. Odečteme-li od registru s obsahem 0 číslo 1, jeho obsah bude FFH. V obou těchto případech bude CY ve stavu log.1.

Nedojde-li při operaci ovlivňující CY k přenosu „9., resp. 17. bitu", je CY ve stavu log.0. Instrukce rotace a posuvu pojednávají CY jako 9. bit akumulátoru. Zevrubně si je probereme v jedné z dalších kapitol.

CY je jediný SI přímo ovlivnitelný dvěma instrukcemi:

SCF *(Set Carry Flag)* nastavuje CY do stavu log.1,
CCF *(Complement Carry Flag)* jej převrací do opačného stavu, než v jakém byl před provedením instrukce. Např. je-li CY = 1, po CCF bude CY = 0, po opětném provedení CCF bude CY = 1.

Zero flag (indikátor nuly) - Z

Jeho logický stav je ovlivňován mnoha instrukcemi. U tohoto SI si musíte zapamatovat, že Z=1, když výsledkem testované operace je nula (mimo dále uvedené výjimky). A naopak - pokud je výsledek operace různý od nuly, je Z=0! Začátečníci si to zpočátku často pletou. Uvědomte si, že Z indikuje nulový výsledek. Stav log.0 indikátoru Z berte jako „Tak o tohle nemám zájem, nic se neděje". Zato log.1 (výsledkem operace je nula) říká: „Pozor, je to tady, zvyšuju napětí!".

Nejen u tohoto indikátoru, ale i u všech ostatních si musíte v tabulkách pečlivě probrat (a při programováni je mít stále po ruce), jaké instrukce ovlivňují které SI, jinými slovy - kdy a čím lze co a jak testovat a co a jak mění stavy kterých SI. I poměrně zkušení programátoři opomíjejí mnohé z poměrně širokého spektra testovacích možností při provádění různých programových operací a zůstávají jen u několika málo základních testů, které si pamatují z doby, kdy se před lety učili programovat. Diky tomuto přístupu se však mohou dopustit i nepříjemných chyb. Jednou z těch častějších je opomenuti toho, že zatímco instrukce snížení obsahu jednoho registru (např. DEC C) ovlivňují Z, pak ty, které snižují obsah párového registru (např. DEC BC), indikátor Z neovlivňují. Chyba nastává tehdy, když programátor za instrukci DEC BC zařadí test nuly, který se samozřejmě nemůže projevit, čímž programátor uvádí sám sebe do nemilého omylu.

Kromě zmíněné instrukce DEC r ovlivňuji indikátor Z např. instrukce INC r a typu BIT (testuje stav jednoho z bitů registru nebo místa v paměti), CP (porovnává obsahy dvou bajtů) atd. Zda je indikátor Z ve stavu log.1 nebo log.0, zjišťujeme zařazením podmínky Z nebo NZ v podmíněných instrukcích skoku, volání a návratu. Přitom si musíme uvědomit, že momentální stav indikátoru odpovídá výsledku poslední z předchozích operací, které jej ovlivňují. Tak vlastně neprovádíme pouhý test stavu indikátoru Z, ale zjišťujeme, zda výsledek takové operace je, či není nulový. Analogicky to platí i u dalších indikátorů.

Sign flag (indikátor znaménka - plus, mínus) – S

Jde o test stavu nejvyššího bitu výsledku operace v rámci dvojkově komplementárních čísel. Když je tento bit ve stavu log. 1, je číslo záporné, a naopak. Indikátor S tedy kopíruje stav tohoto bitu.

Parity/Overflow flag (parita nebo aritmetické přeplnění) - P/V

Při testu parity zjišťujeme, zda je počet bitů log.1 (tedy i bitů log.0 - bitů je přece osm!) sudý nebo lichý. V případě sudé parity je stav indikátoru log.1, v případě liché log.0. Vztahuje se především k logickým operacím.

Aritmetické přeplnění se indikuje tehdy, je-li výsledkem součtu dvou záporných čísel číslo kladné, nebo je-li součet dvou kladných čísel záporný. V tom případě bude stav in-

dikátoru log.1. Pokud jste neporozuměli, vraťte se k výukovému textu o DK číslech. Při programováni si nesmíte plést přeplnění signalizované indikátory P/V a CY. Proveďte si jejich porovnáni a rozdíl si zapamatujte:

```
  1 1 1 1 1 0 1 1      DK reprezentace čísla  - 5
+ 1 1 1 1 0 0 0 0      DK reprezentace čísla -16
  1 1 1 0 1 0 1 1      DK reprezentace čísla -21
```

CY= 1, protože došlo k přenosu z nejvyššího bitu (do „9."). V=0, protože nedošlo k aritmetickému přeplnění (V=1 např. při součtu DK čísel -80 a -70, neboť výsledek přesahuje interval -128..+ 127).

Half-carry a Subtract flags - H a N

To jsou indikátory polovičního přenosu a odečítáni. Oba slouží vnitřnímu systému Z80 a programátorovi jsou nepřístupné (nemůžeme je testovat přímo). Indikátor H nás informuje o tom, že v bajtu došlo k přenosu bitu mezi jeho bity 3 a 4 (v obou směrech). Indikátor N nám oznamuje, že byl proveden (jakýkoli) odečet. Oba SI jsou důležité zvláště při aritmetických operacích s čísly ve tvaru BCD (viz dále).

Bity SI jsou v registru F umístěny takto:

7	6	5	4	3	2	1	0
S	Z	-	H	-	P/V	N	CY

Znak „-" znamená, že bit na jeho místě nemá žádné použití.

V anglicko - českém slovníku výpočetní techniky je slovo *flag* přeloženo i jako příznak. Tento překlad považuji za velmi nepřesný. Příznak je projevem, který nás může vést k indikaci příčiny stavu, příznakem se projevujícím. Má-li nemocný horečku, je tato horečka příznakem nemoci, kterou můžeme indikovat až další postupnou analýzou. Horečka sama je příznakem řady onemocnění. Projev indikátoru registru F s jeho dvěma mezními stavy „buď - anebo" je však naprosto jednoznačnou indikací výsledku operace, k níž se tato indikace přímo váže.

SKOKOVÉ TESTY

U skoků typu JP můžeme testovat všechny čtyři testovatelné SI:

Instrukce	Skok se provede, když:
JP NZ	Z = 0 (výsledek operace není nulový)
JP Z	Z = 1 (výsledek operace je nula)
JP NC	CY = 0 (nedošlo k přeplnění bajtu)
JP C	CY = 1 došlo k přeplnění bajtu, resp. přenosu bitu)

JP PO	P/V = 0 (lichá parita nebo žádné aritmetické přeplněni)
JP PE	P/V = 1 (sudá parita nebo aritm. přeplnění)
JP P	S=0 (DK výsledek je kladný, resp. nezáporný)
JP M	S = 1 DK výsledek je záporný)

Už jsme si řekli, že oproti skokům typu JP nemohou skoky JR testovat všechny SI. Relativní skoky mohou testovat pouze dva z nich - CY a Z: JR NZ, JR Z, JR NC, JR C.

INSTRUKCE DJNZ

Zcela zvláštní skokovou instrukcí je DJNZ. V podstatě jde o instrukci podmíněného relativního skoku s opakováním. Počet opakování skoků se inicializuje registrem B. Například:

```
A100              LD B, 08
A102    ODEČET:   DEC C
A103              DJNZ ODECET
A105              RET
```

Inicializací registru B na adrese A100H je nastaven počet skoků instrukce DJNZ na 8. Kdykoli dojde program na instrukci DJNZ ODECET bude proveden skok na udanou adresu (zde je určena návěštím ODECET). Adresa se stanoví stejně jako u skoků relativních, tedy odchylkou udanou jednobajtovým DK číslem (zde by odchylka byla FDH). Těsně před provedením každého skoku instrukce DJNZ sníží obsah registru B o 1. Skoky se budou provádět, dokud obsah registru B nebude nulový. Z toho vyplývá, že v příkladu bude instrukce DEC C provedena celkem osmkrát (obsah registru C bude snížen o 8). DJNZ můžeme tedy použít vždy, kdykoli potřebujeme provést nějakou operaci vícenásobně.

VOLÁNÍ A NÁVRATY

V přeneseném slova smyslu jde o volání jednotlivých částí „programové divočiny". Tou divočinou jsou samozřejmě míněny podprogramy. Instrukce volání CALL XXXX obsahuje vždy absolutní adresu XXXX a může být buď nepodmíněná, nebo podmíněná (obdoba instrukce typu JP). Touto instrukcí můžeme testovat všechny přístupné indikátory jako u instrukce JP: CALL NZ, CALL Z, CALL NC, CALL C, CALL PO, CALL PE, CALL M, CALL P. Principiálně jsou důsledky testů tytéž jako u podmíněných instrukcí JP. Každá instrukce CALL by měla (ale nutně nemusí) mít někde v programu přiřazenu jednu instrukci návratu RET. Jejich vztah je popsán v textu zabývajícím se programovým řízením (registrem PC).

I instrukce RET může být buď nepodmíněná (tehdy se návrat k posledně provedené instrukci CALL, resp. na adresu danou obsahem spodních dvou adres zásobníku, provede vždy), nebo podmíněná. Stejně jako u instrukcí JP a CALL se testy vztahují ke

všem čtyřem SI: RET NZ, RET Z, RET NC, RET C, RET PO, RET PE, RET M, RET P. Tyto instrukce se provedou jen tehdy, Jsou-li podmínky v nich uvedené splněny.

Nepodmíněné instrukci CALL podobná je instrukce RST *(Restart).* Víme, že těchto speciálních jednobajtových instrukcí je celkem osm. Zapamatujte si, že i tyto instrukce by měly mít někde v programu své dvojče RET. Návrat se provede stejným způsobem jako u instrukcí CALL. Také v tomto případě může být instrukce RET i podmíněná.

Při práci se zásobníkem musíme dát pozor, abychom do něj neuložili takové kvantum adres návratů, že by zásobník začal přepisovat náš program. Připomeňme si opět, že při vlastním programování je třeba věnovat maximální pozornost tomu, aby při odběru adresy návratu instrukce RET odebrala ze zásobníku opravdu tu, kterou odebrat má. Většina počítačů po své inicializaci sama nastavuje adresu zásobníku na některou z nejvyšších adres paměti RAM. Může se ovšem stát, že tomu tak nebude. Pak by se zásobník mohl ocitnout někde uprostřed vašeho programu, který by pochopitelně ve velmi krátké době zkolaboval. Proto je někdy vhodné, když do svého programu zařadíte inicializaci registru SP podle svých potřeb (pozor však na kolaps systému).

A zákon poslední - strukturalizace, strukturalizace a zase strukturalizace!!!

EXPERIMENT č. 1

V něm si probereme funkci instrukce DJNZ.

```
A100  0609          LD B, 09        ;V reg. B počet cyklů smyčky SMYC1
A102  0EFF  SMYC1:  LD C, FFH       ;V reg. C počet cyklů smyčky SMYC2
A104  16FF  SHYC2:  LD D, FFH       ;V reg. D počet cyklů smyčky SMYC3
A106  15    SMYC3:  DEC D           ;Odečítání cyklů SMYC3
A107  20FD          JR NZ, SMYC3    ;Je-li Z = 0, na SMYC3, jinak dál
A109  0D            DEC C           ;Odečítáni cyklů SMYC2
A10A  20F8          JR NZ, SMYC2    ;Je-li Z=0, na SMYC2, jinak dál
A10C  10F4          DJNZ SMYC1      ;Odečítání cyklů SMYC1
A10E  C9            RET             ;Návrat, když obsah reg. B =0
```

KROK 1: Zapište program do počítače a vyzkoušejte jeho funkce s různými obsahy registru B. Jistě si vzpomenete na časovací smyčky z předchozích experimentů. Instrukce DJNZ je pro jejich tvorbu jako stvořená. Proto se velmi často používá v rutinách, kde je časování kritické - např. v částech operačních systémů, které vykonávají funkce SAVE a LOAD.

V našem experimentu dosáhneme změnou obsahu registru B těchto časových výsledků doby běhu celé rutiny:

Registr B	Exekuční čas rutiny
01	0,4420844 sec
09	3,66123772 sec
FFH	105,47 sec

EXPERIMENT č. 2

Ukážeme si první příklad programového použití části programu jako subrutiny, kterou bude rutina z předchozího experimentu.

```
A11E  0603            LD B, 03        ;Počet cyklů smyčky ZPOZD
A120  CD02A1          CALL ZPOZD      ;Volání subrutiny ZPOZD
A123  C9              RET             ;Konečný návrat

A102  0EFF    ZPOZD:  LD C, FFH
A104  16FF    SMYC2:  LD D, FFH
A106  15      SMYC3:  DEC D
A107  20FD            JR NZ, SMYC3
A109  0D              DEC C
A10A  20F8            JR NZ, SMYC2
A10C  10F4            DJNZ ZPOZD
A10E  C9              RET             ;Návrat ze subrutiny
```

KROK 1: Poté, co program provede instrukci volání, přejde programové řízení na subrutinu ZPOZD (registr PC se naplní adresou A102H). Zároveň je do zásobníku přenesena adresa, na niž leží první instrukce za instrukcí voláni CALL ZPOZD. Subrutina se provede. Jakmile registr B dosáhne hodnoty nula, instrukce RET přenese obsah spodních dvou adres zásobníku do registru PC a provede se „zpětný" skok na první instrukci za instrukcí CALL ZPOZD.

V subrutině jsme oproti jejímu zápisu v experimentu č.1 provedli malou změnu. Místo názvu návěští SMYC1 jsme použili název ZPOZD. Důvod pro tento krok je spíše estetický - zároveň však zvyšuje srozumitelnost a přehlednost programu. Je lepší, když ve struktuře programu máme subrutinu s plně sdělným názvem ZPOZD (zpoždění, časová prodleva) než SMYC1 (smyčka 1). Přesnější identifikace funkce rutin jejich výstižným názvem velmi významně usnadňuje orientaci v každém programu. Proto věnujte symbolickému pojmenování programových funkcí náležitou pozornost.

Subrutina plní funkci požadovaného zpracování výstupních parametrů předchozí rutiny (jsou to tzv. přenášené parametry). Jinými slovy - přenášené parametry jsou zpracovávány subrutinou jako její vstupní parametry. Přitom nesmíme zapomenout uschovat na bezpečné místo ty obsahy registrů, které budeme po návratu ze subrutiny ještě potřebovat. Proto je u každé subrutiny důležitý údaj o tom, se kterými registry pracuje. Tak budeme předem vědět, obsahy kterých jsou subrutinou ohroženy a v případě nezbytnosti podnikneme kroky pro jejich přechodné uložení jinam. Nejčastěji se parametry ukládají do zásobníku instrukcí PUSH. Způsobů úschovy je však celá řada. Po návratu ze subrutiny můžeme uložené hodnoty bajtů (i výstupních parametrů) přenést zpět do registrů instrukcí POP či jiným způsobem, jak si ukážeme v příštích experimentech přenosu parametrů do subrutin. Přenos uložených čísel provedeme ihned, jakmile s nimi chceme pracovat, ale i v případě, že by nám v zásobníku či jinde překážely, nebo by jim hrozilo přepsání. Stejně jako vstupní parametry musíme zvážit i výstupní, tedy jaké, jak a proč má subrutina zpracovat, a co s nimi dál.

Obecně platí, že subrutiny musejí splňovat dvě kritéria - funkční a ekonomické. Funkční jsme si probrali v předchozích dvou odstavcích. K ekonomickým patři např. úvaha nad tím, zda zařazení nějaké subrutiny není zbytečným luxusem - zda by jí simulovanou funkci nebylo možné realizovat jinak, nebo ji zahrnout do jiné části programu, sloučit ji s podobnou funkcí jiné subrutiny apod. Ve výsledku jde o šetření pamětí nebo zkrácení prováděcího času celého programu či jeho částí.

Obě tato kritéria se často dostávají do výrazně nepřímé závislosti. V součtu patři prováděcí doba instrukcí CALL a RET v instrukčním souboru Z80 k jedněm z nejdelších. Je to pochopitelné - při provádění CALL XXXX se instrukce dekóduje, reg. PC je naplněn číslem absolutní adresy „skoku", je snížen obsah reg. SP o 2 a vypočtena adresa návratu, která se nakonec ještě přenese do zásobníku. Pak teprve je převedeno řízení programu na adresu XXXX. U RET se přenesou dva spodní bajty zásobníku do PC, zvýší se obsah reg. SP o 2 a provede se „skok" na adresu obsaženou v reg. PC. To vše samozřejmě zabere dost času. Proto je v některých časově kritických operacích lepší subrutiny nesestavovat, i když je to na úkor rozsahu volné paměti. Zde by se jako určité východisko jevilo použití krátkých instrukcí RST. Tyto instrukce jsou však takřka vždy využity výrobcem počítače k obslužným operacím jeho systému.

KROK 2: Krokujte programem a zaměřte svou pozornost na registry PC, SP a zásobník před provedením instrukci CALL a RET a po něm. Uvědomte si příčiny změn, které v nich probíhají. V případě nejasnosti se vraťte k předchozím odstavcům.

Nyní se pokuste analyzovat celý program z hlediska obou výše uvedených kritérií. Pokud jde o kritérium ekonomické, časově i prostorově je na tom užití subrutiny v našem programu velmi bledě. Proč? Protože subrutina je užita jen jednou. Čas provedení celého programu se tak zbytečně prodlužuje o instrukce CALL a RET, které navíc zabírají 4 bajty pamětí. Pro tento případ by bylo mnohem výhodnější funkci subrutiny zařadit přímo do hlavní rutiny:

```
        LD B, 03
ZPOZD:  LD C, FFH
SMYC2:  LD D, FFH
SMYC3:  DEC D
        JR NZ , SMYC3
        DEC C
        JR NZ , SMYC2
        DJNZ ZPOZD
        LD A,00
        RET
```

Tento program má 17 bajtů, zatímco předchozí jich měl 22. Ušetřili jsme tedy přibližně čtvrtinu bajtů! Tím se nám potvrdilo, že zařazení jakékoli funkce části programu do přímé linky jeho vývoje vždy přináší vyšší rychlost provedení, než je tomu při užití subrutin. Chceme-li však subrutinu programu použít vícenásobně, přimhouříme oko nad nějakým zlomkem vteřiny navíc a subrutinu do programu zařadíme. Tímto strukturalizačním krokem dosáhneme mnohem větších výhod, které s sebou dobře sestavená programová struktura přináší. Z hlediska funkčnosti jsou obě varianty v pořádku.

EXPERIMENT č.3

Kromě jiného si ukážeme použití instrukce RST XX.

```
; Hlavní rutina:
;
A133 010001          LD BC, 0100H    ;Čítač počtu bajtů pro vynulováni
A136 2100A4          LD HL, A400H    ;1.adr. nulovaného bloku bajtů
A139 D7              RST 10H         ;Volání adresy 16
A13A C9              RET             ;Návrat
;
; subrutina:
;
0010 F5              PUSH AF         ;Úschova obsahů „ohrožených"
0011 C5              PUSH BC         ;registrů do zásobníku
0012 D5              PUSH DE
0013 E5              PUSH HL
0014 3600            LD (HL),00      ;Naplnění 1. adresy bajtem nula
0016 54              LD D,H          ;Přenos obsahu reg. HL do reg. C
0017 50              LD E, L
0018 13              INC DE          ;Přičtení 1 k DE
0019 EDB0            LDIR            ;Vynulování celého bloku bajtů
001B 0603            LD B, 03        ;Časová konstanta prodlení
001D CD02A1          CALL ZPOZD      ;Voláni subrutiny ZPOZD
0020 E1              POP HL          ;"Vyskladnění" uschovaných ob-
0021 C1              POP BC          ;sahů registrů ze zásobníku
0022 01              POP DE          ;zpět do registrů
0023 F1              POP AF
0024 C9              RET             ;Návrat do hlavní rutiny (na
                                     ;adresu A13AH)
```

KROK 1: Má-li váš počítač volnou paměť od adresy 0010H, program zapište a zkontrolujte správnost zápisu. Subrutinu ZPOZD použijte z minulého experimentu.

KROK 2: Uvedený program nuluje určený počet bajtů (zde 101H) v určeném bloku sousedících adres paměti (zde od adresy A400H). Podobnou funkci obsahuje řada systémových i užitkových programů.

Hlavní program používá registry BC a HL pro přenos parametrů do subrutiny umístěné od adresy 10H. BC obsahuje počet adres, jejichž obsah je určen k vynulování; v HL je první adresa tohoto bloku adres. V subrutině je uložena jedna nula na první adresu bloku instrukcí LD (HL),00. Funkce LDIR pak okopíruje nulový obsah této adresy do všech dalších 100H adres bloku. Zcela první přenos proběhne tak, že nula z první adresy bloku (HL) se přenese na adresu (DE), která je o 1 vyšší. Obsahy HL a DE se pak BC+1 krát (na to pozor!) zvýší o 1, přičemž proběhne BC přenosů z adresy s nulovým bajtem (HL) na adresu o 1 vyšší (DE). Ve výsledku budou mít všechny adresy bloku nu-

lový obsah. Subrutina je volána jednobajtovou instrukcí RST 10H (stejně tak bychom ji mohli volat instrukcí CALL 0010H, ovšem ta je tříbajtová).

Na začátku subrutiny s blokovým přenosem dat LDIR jsou instrukce PUSH, které uschovávají obsahy všech párových registrů a reg. A a F do zásobníku. Tak jejich původní hodnoty zůstanou nedotčeny. Pokud by některá ze subrutin hlavního programu nepoužívala některý z párových nebo AF registrů, nemuseli bychom jejich obsah uschovávat. Voláni subrutiny ZPOZD je zde na ukázku toho, že i ze subrutin můžeme volat další subrutiny a z nich zase další a další. Toto vnořování subrutin je velmi náročné z hlediska uhlídání adekvátních adres návratů v zásobníku i orientace ve struktuře programu. Je však velmi silnou zbraní programátora. Subrutiny dokonce mohou volat samy sebe - tento způsob volání se jmenuje rekurzívní. Jde o užití podmíněné instrukce CALL na způsob instrukce DJNZ. Pokud použijeme nepodmíněnou instrukci CALL, musí být v subrutině umístěna nějaká jiná (splnitelná) podmínka, která nás z ní vyvede.

Vedle vývojového diagramu je velmi užitečnou orientační pomůckou paměťová mapa programu. Můžete si ji doplnit řadou vlastních orientačních poznámek. Pro náš experiment vypadá takto:

0000 000F	
0010 0024	Subrutina
0025 A101	
A102 A10E	Subrutina ZPOZD
A10F A132	
A133 A13A	Hlavní program
A13B A3FF	
A400 I A500	Data
FFA0 FFBB	Zásobník

EXPERIMENT č. 4

Na čtyřech příkladech si ukážeme čtyři základní způsoby přenosu parametrů do subrutin. Všechny programy jsou funkčně shodné s programem předchozím (subrutina ZPOZD je vynechána)

```
;_____________________________________________________________________
;
;Technika 1. - přenos parametrů prostřednictvím registrů

A203  010001              LD BC,0100H    ;Počet adres k vynulování
A206  2100A4              LD HL, A400H   ;1. adresa bloku
A209  CD10A2              CALL NULA1     ;Volání subrutiny NULA1
A10C  C9                  RET
;
;Subrutina NULA 1
;
A210  3600      NULA1:    LD(HL), 00     ;Nula na 1. adresu bloku
A212  54                  LD D,H         ;Přenos HL do DE
A213  5D                  LD E, L
A214  13                  INC DE
A215  ED80                LDIR           ;Přenos nul na adresy bloku
A217  C9                  RET

;_____________________________________________________________________
;
;Technika 2. – přenos parametrů prostřednictvím zásobníku
;
A223  010001              LD BC, 0100H   ;Počet adres k vynulováni
A226  2100A4              LD HL, A400H   ;1. adresa bloku
A229  C5                  PUSH BC        ;Uložení parametrů (obsahu
A22A  E5                  PUSH HL        ;registrů) do zásobníku
A22B  CD10A2              CALL NULA2     ;Volání subrutiny NULA2
A10C  C9                  RET
;
; Subrutina NULA2
;
A231  D1        NULA2:    POP DE         ;Adresa návratu do DE
A232  E1                  POP HL         ;Přenos parametrů ze zásobníku
A233  C1                  POP BC
A234  D5                  PUSH DE        ;Adresa návratu zpět do zásob.
A235  3600                LD (HL), 00
A237  54                  LD D, H
A238  5D                  LD E, L
A239  13                  INC DE
A23A  EDB0                LDIR
A23C  C9                  RET
```

```
;_____________________________________________________________________
;
; Technika 3. – přenos parametrů prostřednictvím bloku paměti
;
A243  010001              LD BC, 0100H
A246  2100A4              LD HL,A400H
A249  ED4300A8            LD (A800H),BC   ;Parametry na adresy A800H až
A24D  2202A8              LD (A802H),HL   ;A803H
A250  CD56A2              CALL NULA3
A253  C9                  RET
;
; Subrutina NULA3
;
A256  ED4800A8 NULA3:     LD BC, (A800H)  ;Parametry zpět do registrů
A25A  2A02A8              LD HL, (A802H)  ;z adres jejich uložení
A250  3600                LD (HL),00
A25F  54                  LD D,H
A260  5D                  LD E, L
A261  13                  INC DE
A262  EDB0                LDIR
A264  C9                  RET
;_____________________________________________________________________
;
; Technika 4. – přenos parametrů prostřednictvím adres následujících těsně
;                za instrukcí volání subrutin
;
A268  CD72A2              CALL NULA4
A26B  0001                DEFW 0100H      ;Uložení parametrů jako defino-
A26D  00A4                DEFW A400H      ;vaných slov (DEFined Words)
A26F  C9                  RET
;
; Subrutina NULA4
;
A272  DDE1       NULA4:   POP IX          ;Adresa „návratu" do IX
A274  DD4E00              LD C, (IX+00)   ;Přenos parametrů z adres
A277  DD4601              LD B, (IX+01)   ;A26BH až A26EH do reg. BC a HL
A27A  DD6E02              LD L, (IX+02)
A27D  DD6603              LD H, (IX+03)
A280  110400              LD DE, 0004     ;"Odchylka" adresy návratu
A283  DD19                ADD IX, DE      ;Nastavení skutečné adr. návratu
A285  DDE5                PUSH IX         ;Uschování adr. návratu do zás.
A287  3600                LD (HL), 00
A289  54                  LD D, H
A28A  5D                  LD E, L
A28B  13                  INC DE
A28C  EDB0                LDIR
A28E  C9                  RET             ;Návrat na adresu A26FH
```

KROK 1: Zapište všechny programy do počítače a zkontrolujte zápis.

KROK 2: Přestože výsledkem činnosti všech čtyř programů je totéž, v lecčem se liší. Uvedené základní techniky přenosu parametrů do subrutin však neznamenají, že nelze vymyslet ještě nějaké další.

KROK 3: Nyní si porovnáme všech pět technik z hlediska ekonomického. Tabulka ukazuje, kolik adres paměti jednotlivé programy zabírají.

Technika	Hlavní rutina	Subrutina	DEFW	Součet bajtů
1	15	8	0	23
2	17	12	0	29
3	22	15	0	37
4	9	29	4	42

I když technika 4 vypadá na první pohled nejhůř, je nutno si uvědomit, že pro volání a nastavení parametrů před jejich přenosem do subrutiny spotřebuje vždy jen 7 bajtů. Techniky 1, 2 a 3 spotřebují bajtů vždy 9. Z toho vyplývá jeden zajímavý poznatek - čím vícekrát budeme subrutinu sestavenou technikou 4 volat, tím více se bude snižovat rozdíl v časové ztrátě jejího provedení. A po překročení určitého počtu volání bude součet časů provedení nakonec nižší než u kratších subrutin vázaných na techniky, jež pro volání a nastavení parametrů vyžaduji více bajtů. V našem případě je však subrutina se 7-bajtovým voláním a přenosem příliš dlouhá, proto nemůže porazit ani jednu ze tří předchozích.

Pro přenos mnohem většího počtu parametrů, než jsou zde přenášené čtyři, se jako zcela nevhodná jeví technika 1. Ani 2. by nám moc nepomohla. O dost lépe už je na tom technika 3. S malými úpravami subrutiny by byla nejvhodnější technika 4, v níž jsou bajty uloženy přímo na adresy, z nichž si je subrutina odebírá. Ovšem pro přenos většího počtu parametrů by musela být rovněž trochu upravena. Nejlépe by však bylo pro tento účel sestavit zcela nový program. V této knížce se s takovými ještě setkáte. Uvedené programy se tedy vyznačují spíše jistým stupněm jednoúčelovosti.

Jeden důležitý dodatek ke všem programům v experimentech uvedeným. Na jejich konci je vždy umístěna instrukce RET pro konečný návrat k instrukci jejich volání. Pokud program mění obsah SP, je nutné na jeho vstupu někam uschovat původní hodnotu SP a před konečným návratem ji opět obnovit. U některých počítačů je možno využít i přímého skoku do jejich operačního systému (např. interpretu Basicu), který převezme řízeni programu, aniž by cokoli zkolabovalo. U většiny našich programů SP nemusíme předem nastavovat, protože bývá automaticky nastaven systémem počítače. V každém případě - pozor na změny obsahu reg. SP.

Čtyřmi ukázkami provedení téhož různými způsoby jste zaklepali na vstupní bránu do pestrého světa programovacích technik. Je to svět bez hranic, říše nekonečné fantazie. Z jejího bohatství nelze čerpat bez širokého spektra základních znalostí, které však zůstávají přece jen pouhým nástrojem. Virtuozita přichází až tehdy, kdy nástroj, veden fantazií tvůrce, jako by hrál sám. Protože stojíte teprve na začátku dlouhé cesty, pohrajte si s uvedenými programy především pomocí krokování, abyste je plně pochopili.

EXPERIMENT č. 5

Následující program demonstruje jednu z mnoha možností užití datové oblasti definovaných bajtů. V tomto případě jde o sestaveni a využití oblasti dat jako tabulky adres skokových instrukcí.

```
A900  61                DEFB 61H        ;1. bajt porovnání pro případný
A901  41A9              DEFW A941H      ;skok na tuto 1. adresu - JP (HL)
A903  62                DEFB 62H        ;2. bajt... atd.
A904  45A9              DEFW A945H
A906  63                DEFB 63H
A907  50A9              DEFW A950H
A909  64                DEFB 64H
A90A  55A9              DEFW A955H
A90C  00                DEFB 00         ;Indikace konce tabulky
A915  21FDA8    START:  LD HL, A8FDH    ;Začátek programu
A918  23        DALSI:  INC HL
A919  23                INC HL
A91A  23                INC HL
A91B  7E                LD A, (HL)      ;Obsah adr.(HL) do reg. A
A91C  B7                OR A            ;Log. součet obsahu reg. A se sebou
A91D  C8                RET Z           ;Je obsah A = 0? ANO, pak návrat
A920  B8                CP B            ;NE, pak porovnání reg. B s reg. A
A921  20F5              JR NZ, DALSI    ;Není-li B = A, pro další DEFB
A923  23                INC HL          ;Je-li B=A, z další adr. (HL) pře-
A924  5E                LD A, (HL)      ;nes nižší bajt adresy skoku do A
A925  23                INC HL          ;a z další adr. (HL) přenes
A926  66                LD H, (HL)      ;vyšší bajt adr. skoku do reg. H
A927  6F                LD L, A         ;Převeď obsah reg. A do L
A928  E9                JP (HL)         ;Proveď skok na nalezenou adresu
```

KROK 1: V oblasti dat se nám vedle termínu DEFW objevil nový - DEFB. DEFW *(DEFine Word)* definuje slovo, které obsahuje dva bajty. DEFB *(DEFine Byte)* definuje bajt. Jde čistě jen o symboliku doplňkových funkci generátoru strojového kódu, která s vlastní mnemonikou Z80 nemá nic společného. V assemblerových programech budeme tyto tzv. pseudoinstrukce (též zvané pseudooperační kódy) často používat, abychom se vyznali v tom, co jsou instrukce a co data. Rozdíl mezi instrukcemi a daty jsme si již vysvětlili. Při programování v assembleru nám některé generátory umožňují používat i jiné pseudoinstrukce. Pro definování řetězce (např. textu) bývají vybaveny pseudoinstrukcí DEFM, rozsahu vyhrazené paměti DEFS a dalšími. V podstatě jde vždy jen o uložení bajtů určité hodnoty na místa paměti, která v souhrnu tvoři oblast dat. Ať už se označují jakkoli, neuděláme chybu, když jim všem budeme říkat prostě definované bajty a jimi okupovanému rozsahu paměti oblast dat.

Instrukce typu CP je pro nás ještě neznámá, řada na ni dojde později. Prozatím si řekneme jen tolik, že porovnává obsah jí určeného registru (nebo adresy) s obsahem

akumulátoru, přičemž obsahy všech zúčastněných prvků zůstávají beze změny. V našem případě jde o odečet obsahu registru B od obsahu akumulátoru instrukcí CP B, ale s tím, že výsledek odečtu se nikam neukládá, jsou jím pouze ovlivňovány indikátory registru F. V programu za touto instrukcí testujeme stav indikátoru Z. Zapište program do počítače a zkontrolujte správnost zápisu.

KROK 2: Datovou oblast uvedeného programu si nazveme skoková tabulka. Představte si rutinu, která zjišťuje kód stisknutého tlačítka klávesnice. Další chod programu bude závislý na tom, které tlačítko bude stisknuto. Je samozřejmé, že taková rutina nebude mít jeden, ale celou řadu výstupů. Pro řešení takové situace je velmi výhodná konstrukce datové tabulky obsahující adresy skoků do různých částí programu. Rutina na základě určitého testu provede výběr tabulkové adresy, na niž bude převedeno programové řízení. Způsobů konstrukcí takových tabulek je více. Podíváme se na jeden z nich.

Před vstupem do uvedené rutiny obsahuje reg. B bajt který je výsledkem nějaké předchozí operace. Jeho obsah určuje, jaká adresa skoku bude z tabulky vybrána, tedy jaký skok bude následně proveden. Tak např. pro volbu skoku na adresu A950H musí být vstupní obsah registru B v předchozí části programu 63H. Instrukcí CP B se zjistí, kdy tuto hodnotu bude obsahovat i akumulátor. V tom případě bude ignorována instrukce JR NZ, DALSI a registr HL bude v závěru obsahovat adresu skoku.

Pár instrukcí LD A,(HL) a OR A zjišťuje, zda je registr A (tedy i obsah adresy (HL)) nulový. Nulu v uvedené tabulce používáme pro indikaci jejího konce. Tento bajt se přeneseně označuje jako nárazník. Pokud by některý z DEFB byl rovněž nulový, museli bychom pro nárazník zvolit jiný bajt, a to takový, který neobsahuje ani jeden z předchozích DEFB.

Pro demonstraci konstrukce a využití datových tabulek je uvedený případ dostatečně ilustrativní. Z hlediska funkčního by však program bylo možno sestavit o něco lépe, ostatně jako téměř vždy. V tomto případě by stálo za úvahu, zda by nebylo lepší instrukce JP s definovanými absolutními adresami zařadit do jedné posloupnosti a vybírat je jednou instrukcí JR XX, umístěnou v čele posloupnosti. Protože instrukce JP jsou tříbajtové, musel by být DEFB před jeho uložením do bajtu odchylky XX vynásoben třemi. Instrukce JR XX by pak vždy provedla skok na vybranou instrukci JP, která by byla následně provedena. Mj. je to i názorný příklad techniky *self-modification* (v průběhu programu měníme přímo bajt instrukce). Variant je samozřejmě celá řada. Nevýhodou prováděného testu (CP B) je nutnost projít tabulkou vždy od jejího začátku až do nálezu DEFB, shodného s obsahem registru B. Tak je doba provedení závislá na tom, jak daleko od začátku tabulky hledaný DEFB je.

Podobnou tabulku lze využít pro přepočty různých hodnot, konverze kódů atd. Alternativy využití datových oblastí si ukážeme i v dalších experimentech.

KAPITOLA 4
LOGICKÉ INSTRUKCE

Tyto instrukce mají svůj význam v mnoha operacích, které provádíme s bity bajtů. Jejich použitím můžeme mnohdy podstatně zkrátit program a vyhovět všem programovým kritériím. Proto je zvládnutí těchto instrukcí a hlavně možností jejich uplatnění v programu naprosto nutným předpokladem programování ve strojovém kódu. Bohužel, mnozí programátoři si často neuvědomují aplikační možnosti logických instrukcí. Operace pak provádějí zbytečným nánosem „substitučních" instrukcí podle známého hesla „proč se drbat takhle, když to jde takhle". Způsobů využití logických instrukcí je nepřeberné množství. Proto zde uvedené experimenty nemohou ani zdaleka ukázat všechny možnosti jejich uplatnění.

CO JE LOGICKÁ INSTRUKCE?

Tento typ instrukce provádí logické operace, s nimiž jste se seznámili při výuce základů Booleovy algebry v první části knihy. Všechny logické instrukce provádějí operace mezi akumulátorem a zvoleným registrem nebo obsahem místa paměti nebo určeným číslem. Výsledek operace se vždy objeví v reg. A. Druhý operand zůstává nezměněn. V instrukčním souboru Z80 jsou implementovány čtyři logické operace: AND, OR, XOR a NOT.

AND - logický součin
OR - logický součet
XOR - nonekvivalence, též zvaná logický rozdíl
NOT - inverze

Operace NOT je realizována instrukcí CPL a speciální instrukcí CCF. Ostatní instrukce jsou stejnojmenné.

Multibitové operace

Veškeré logické operace mezi bajty je třeba chápat jako logické operace mezi jejich jednotlivými bity se stejným pořadovým číslem (třeba bit 3 akumulátoru provádí logickou operaci vždy s bitem 3 druhého operandu). Čili logické operace jsou souběžně provedenými logickými operacemi mezi páry jednotlivých bitů. Např. výsledek logické operace AND B s níže uvedenými obsahy registrů bude takovýto:

reg. A	11110000
reg. B	00111100
A AND B	00110000

Po operaci OR B bude výsledek:

```
  11110000
  00111100
  --------
  11111100
```

A po XOR B:

```
  11110000
  00111100
  --------
  11001100
```

SKUPINA LOGICKÝCH INSTRUKCÍ Z80

Při seznamování s nimi věnujte zvýšenou pozornost způsobu, jakým ovlivňují jednotlivé indikátory registru F. Na tom se do značné míry zakládá i efektivita jejich využití

Komplementace CY - CCF

CCF provede jedničkovou komplementaci (inverzi) indikátoru přenosu CY, tedy operaci NOT CY. Připomeňme si, že v instrukčním souboru Z80 je ještě jedna instrukce přímo ovlivňující CY - SCF. Jí nastavíme CY na log.1. Firma Zilog obě instrukce řadí do skupiny instrukcí řízení mikroprocesoru.

Komplementace akumulátoru - CPL; NEG

Jak uvedeno výše, operace NOT provádí jedničkovou komplementaci. CPL ovlivňuje obsah akumulátoru, je ekvivalentem operace NOT A. Kromě N a H jiné indikátory reg. F neovlivňuje.

Reg. A před CPL	10111010
Reg. A po CPL	01000101

Užitečnou aplikací instrukce CPL je programové převedení obsahu akumulátoru na 8-bitové DK číslo:

```
CPL
INC A
```

Pro uvedený postup dvojkové komplementace akumulátoru má Z80 instrukci NEG, která provede DK naráz. Rozdíl mezi oběma způsoby provedení DK obsahu reg. A je i v chování SI.

Instrukce AND

Jako všechny logické instrukce i tato operuje s akumulátorem. Instrukční soubor Z80 obsahuje 11 různých instrukcí typu AND. Po provedení instrukce AND se indikátor přenosu CY vždy nastaví na nulu. Indikátory Z a S jsou přímo ovlivněny výsledkem logické operace. Indikátor parity je log.1 při sudé paritě, log.0 při liché. Podstatu funkce AND si ukážeme na příkladu instrukce AND B (na stavu SI před operací nezáleží):

	Akumulátor	S	Z	P/V
Reg. A před AND B	11001100			
Reg. B	10001011			
Reg. A po AND B	10001000	1	0	1

Protože při funkci AND je logický součin nul roven nule a logický součin jedniček roven jedné, mohlo by se zdát užití instrukce AND A naprosto beze smyslu - výsledkem této operace je původní obsah akumulátoru (čili se nic nestane). Operace AND A má však několik velmi významných pomocných funkcí. Jejím provedením zjistíme, zda je obsah akumulátoru nulový (jen tehdy bude Z = 1), obsahuje-li akumulátor před provedením AND A záporné číslo (jen tehdy bude S = 1) a konečně touto operací nulujeme CY, aniž bychom se dotkli obsahu reg. A.

Další užitečnou funkcí operace AND obecně je její využití při tzv. maskování. Při této logické technice jsou určité bity multibitového čísla vynulovány a ostatní zachovány. Maskováním např. vynulujeme ty bity bajtu, které nemají být v nějaké operaci aktivní, nebo naopak. To záleží přímo na situaci v programovém místě maskování. Například chceme zjistit log. stav nejnižšího bitu (tedy bitu 0) z adresy ABCDH:

```
LD A, (ABCDH)
AND 01
```

Ať je obsah adresy jakýkoli, maska 01 operace AND vynuluje všechny bity akumulátoru v rozsahu bit 1 až 7. Hodnota bitu 0 zůstane zachována. V případě, že je bit 0 = 1, pak i obsah reg. A=1 a Z = 0. Když bude bit 0 = 0, i obsah reg. A = 0 a Z=1. Takovouto maskou a zjištěním stavu Z můžeme testovat stav jakéhokoli bitu jakéhokoli bajtu.

Velmi často se maskování používá při přenosu dat mezi počítačem a externími zařízeními, resp. při programovém ovládání jakéhokoli hardwaru. Často se při něm setkáme s tím, že na paralelní datové sběrnici ovlivňuje každý bit (nebo jejich skupiny) některou z více funkcí hardwaru. Pokud potřebujeme vymezit jen jednu z nich, použijeme masku. Tak např. požadujeme, aby se vykonala jen určitá funkce, ovlivňovaná čtveřicí nižších bitů (bity 0-3). Ostatní funkce ovlivňované zbylými čtyřmi bity mají zůstat neaktivní, nulové. Dosáhneme toho maskou:

AND 0FH

Binárně kódová reprezentace bajtu 0FH je 00001111. Všechny 4 nižší výstupní linky pak zůstanou v předem definovaném stavu, ale vyšší 4 linky budou nulové.

Instrukce XOR

Tato instrukce ovlivňuje SI reg. F stejně jako AND a v instrukčním souboru Z80 má rovněž 11 variant. Funkci XOR si představíme instrukcí XOR (HL), která provádí operaci XOR obsahu adresy (HL) s obsahem akumulátoru:

	Akumulátor	S	Z	P/V
Reg. A před XOR (HL)	10010001			
Obsah adresy (HL)	00110000			
Reg. A po XOR (HL)	10100001	1	0	0

Podobně jako operace AND i XOR má své zvláštní stavy. Např. provedením operace XOR 00 zůstane stav reg. A nezměněn, ale všechny SI budou patřičně ovlivněny. Takže Chceme-li zjistit, zda je obsah jakéhokoli bajtu nulový či nenulový, kladný či záporný nebo zda je jeho parita sudá či lichá, použijeme XOR 00.

Instrukce XOR A nuluje akumulátor s následnou indikací Z=1. Jednobajtové instrukce XOR A se v programech používá velmi často, protože je kratší i rychlejší než dvoubajtové LD A,00.

XOR FFH má stejný účinek jako CPL, ale s tím rozdílem, že ovlivňuje všechny SI, zatímco CPL ani jeden ze čtyř testovatelných.

Instrukce OR

Všech 11 variant instrukci OR ovlivňuje všechny SI stejně jako AND a XOR. Funkci OR si ozřejmíme na příkladu OR (IX + 20H). Provedeme tak operaci OR mezi akumulátorem a obsahem adresy o 32 vyšší, než je obsah reg. IX:

	Akumulátor	S	Z	P/V
Reg. A před OR (IX+20H)	00000011			
Obsah adresy (IX+20H)	00110010			
Reg. A po OR (IX + 20H)	00110011	0	0	1

Funkci OR A (resp. OR 00) lze použit pro indikaci toho, zda je obsah akumulátoru nulový. S tímto použitím instrukce OR jsme se již setkali při testu, zda je obsah párového registru roven nule. Toto použití funkce OR je velmi časté, proto si je zapamatujte:

```
LD A, B
OR C
```

Tuto testovací kombinaci nejenže můžeme, ale přímo musíme použít, kdykoli potřebujeme zjistit, zda je obsah kteréhokoli párového registru nulový po jeho dekrementaci (resp. inkrementaci) instrukcí DEC rr (resp. INC rr). Oproti 8-bitovým instrukcím DEC r a INC r 16-bitové instrukce DEC rr a INC rr žádný SI neovlivňují. Pokud tedy oba párové registry budou nulové, bude i výsledek uvedené operace nulový a indikátor Z = 1 nám tento výsledek jasně oznámí. Samozřejmě, že takto lze testovat obsah jakýchkoli dvou 8-bitových registrů kdykoli, tedy nejen pro uvedený případ.

Protože instrukce AND, XOR i OR automaticky nulují indikátor CY, je možno kteroukoli z nich použít pro tento účel. Použijeme samozřejmě takovou, která momentálně nebude mít žádný negativní vliv na průběh našeho programu.

V konečném souhrnu si ještě vyjmenujme logické instrukce, jimiž můžeme testovat nulový obsah akumulátoru: AND A, XOR 00, OR A, OR 00.

EXPERIMENT č. 1

Operace AND mezi obsahy registrů BC a DE s uložením výsledku do reg. HL.

```
A1FA  01MMMM          LD BC, MMMM     Inicializace obsahu BC číslem MMMM
A1F0  11NNNN          LD DE, NNNN     ;Inicializace obsahu DE číslem NNNN
A200  78      AND 16: LD A, B         ;Přenos B do A
A201  A2              AND D           ;Operace B AND D
A202  67              LD H, A         ;Výsledek do H
A203  79              LD A, C         ;Přenos C do A
A204  A3      AND E     ;Operace C AND E
A205  6F              LD L, A         ;Výsledek do L
A206  84              OR H            ;Je-li výsledek = 0, pak ať Z = 1
A207  C9              RET
```

KROK 1: Zkontrolujte správnost zápisu programu do počítače. Proveďte buď přímou inicializaci reg. BC a DE pomocí monitoru (v tom případě vynechte instrukce na adresách A1FAH a A1FDH), nebo obsahy registrů zapište přímo na adresy čísel MMMM a NNNN. Nyní již umíte přenášet data i mezi paměťovými adresami - co kdybyste zkusili inicializovat reg. BC a DE jinak? Zkoušejte, tvořte, uvažujte nad možností řešit programy a jejich části i jinak, než je v textu uvedeno.

Krokujte programem s různými vstupními hodnotami reg. BC a DE:

1.	BC 00110011 11001001	(=33C9H)
	DE 11110000 01110011	(=F073H)
	HL	(zkuste vždy určit předem)
2.	BC 10100101 10100101	(=A5A5H)
	DE 11001100 01011111	(=CC5FH)
	HL	

3. BC 11111111 00000000 (=FF00H)
DE 00000000 11111111 (=00FFH)
HL

Rovněž předem určete závěrečné stavy všech SI registru F. Spuštěním programu se přesvědčte o správnosti svých výpočtů. Výsledky operací AND v reg. HL a konečné stavy SI v pořadí S, Z, P/V a CY by měly být tyto:

1. 3041H 0010 2. 8405H 1000 3. 0000H 0110

EXPERIMENT č.2

Následující program zjišťuje, která z připojených periferních zařízení změnila svůj předchozí stav. Jinými slovy - která z nich přešla ze stavu zapnuto do vypnuto (on — off) nebo naopak z vypnuto do zapnuto (off — on). Jde tedy o příjem dat z periférie (z jejího pohledu výstupních) do počítače (z jeho pohledu vstupních).

Vstup je paralelní, 8-bitový. Na výstupu periferního zařízení použijeme diody LED, jejichž svit budou snímat fotodiody na vstupu počítače. Je-li úroveň bitu vstupního bajtu log.1 (fotodioda indikuje světlo), periferní zařízení je zapnuto, a naopak. K počítači však nemusíme nic připojovat. Připojení diod (hodnotu vstupního bajtu) budeme simulovat inicializací reg. B a reg. A.

```
A100  0688    LD B, 88H     ;Simulační bajt předchozího stavu
A102  3E09    D A, 09       ;Simulační bajt momentálního stavu
A104  4F      D C, A        ;Momentální stav zařízení do reg. C
A105  A8      OR B          ;Bit log.1 po XOR znamená změnu
A106  57      LD D, A       ;Výsledek operace XOR B do D
A107  A0      AND B         ;Bit log.1 znamená změnu on — off
A108  67      LD H, A       ;Výsledek operace AND B do H
A109  2F      CPL           ;"NOT A" — převrácení stavů bitů
A10A  A2      AND D         ;Bit log.1 znamená změnu off — on
A10B  6F      LD L, A       ;Výsledek operace AND D do L
A01C  C9      RET
```

KROK 1: Proveďte kontrolu správnosti zápisu programu do počítače.

KROK 2: Zda jednotlivé diody prošly změnou on — off nebo off — on (či zůstaly beze změny), zjišťujeme postupně:

1. Je-li bit N (N je v intervalu 0..7) registru D ve stavu log.1, pak v N-té diodě došlo ke změně stavu.
2. Je-li bit N registru L ve stavu log.1, pak N-tá dioda prošla změnou on — off (tedy zhasla).

3. Je-li bit N registru H ve stavu log.1, pak N-tá dioda prošla změnou off - n (rozsvítila se).
4. Je-li bit N registru L ve stavu log.0, pak se stav N-té diody nezměnil (svítí i nadále).
5. Je-li bit N registru H ve stavu log.0, pak se stav N-té diody nezměnil (nesvítí i nadále).

KROK 3: Pro absolutní pochopení bitových změn si projděte následující znázornění všech čtyř logických operaci programu:

Předchozí, starý stav vstupního bajtu v reg. B je 10001000. Jeho aktualizovaný, nový stav v reg. A je 00001001.

Instrukce XOR B:	10001000	starý stav
	00001001	nový stav
	10000001	log.1 = změněné stavy
Instrukce AND B:	10001000	starý stav log.1 = změněné stavy
	10000001	
	10000000	log.1 = změna on — off
Instrukce CPL:	10000000	výsledek AND B
	01111111	
Instrukce AND D	01111111	výsledek minulé CPL
	10000001	výsledek XOR B (viz výše)
	00000001	log.1 = off - on, 0 = beze změny

Z toho vyplývá, že zhasla dioda LED 7, rozsvítila se LED 0, zatímco ostatní zůstaly v předchozím stavu (tzn. že LED 3 svítí nadále a LED 1, 2, 4, 5, 6 zůstaly zhasnuté). Nyní tedy svití diody LED 0 a LED 3, ostatní ne, což je potvrzeno bajtem aktualizovaného stavu na vstupu (obsah reg. A). Mimochodem - při výuce stavby hardwaru a jeho programování jsou takováto cvičení se skutečně připojenými diodami LED tou nejnázornější a metodicky nejefektivnější pomůckou

KROK 4: Proč je na adrese A104H přenesen obsah reg. B do reg. C? Pokud bychom chtěli aktivovat tento program jen jednou, byla by instrukce LD C, A zbytečná. Pokud však budeme chtít pro indikaci stavu připojené periférie využít tento program vícekrát, musíme aktualizovaný bajt uschovat, protože pro příště bude bajtem stavu předchozího Tzn. že před dalším vstupem do programu musíme bajt z reg. C přenést *do* reg. B, čímž jej inicializujeme hodnotou předchozího stavu, a obsah reg. A bude aktualizován stavem vstupu. Zamyslete se nad tím, jak byste z uvedeného programu vytvořili subrutinu, která by testovala změny stavu periférie (předávání vstupních a výstupních parametrů, úschova registrů atd.).

CVIČENÍ

Operace Booleovy algebry si přímo vyžadují jejich procvičení. Proto je v žádném případě nevynechte! Správné odpovědi jsou uvedeny za poslední úlohou.

1. Určete výsledky uvedených operací (u HD čísel proveďte jejich převod na binární):

a.	11001011	AND	01011010	e.	CCH	AND	0BH
b.	00100000	OR	11011111	f.	A6H	XOR	80H
c.	00100000	ANO	11011111	g.	37H	XOR	04H
d.	10101010	XOR	10100100	h.	49H	AND	1BH

2. Opět si představte osm detekčních snímačů svitu diod LED 0 až LED 7. Snímače jsou připojeny na paralelní 8-bitový vstup. Budeme zjišťovat, které diody svítí (adekvátní bit bude ve stavu log.1) pro různé hodnoty vstupních bajtů:

a.	55H	f.	30H
b.	40H	g.	06H
c.	64H	h.	C0H
d.	20H	i.	01H
e.	02H	i.	28H

Převody HD čísel na binární si proveďte, i když je to činnost poněkud zdlouhavá. Je nutné, abyste se s ní dobře obeznámili. Když na potřebu převodu při programování narazíte, musíte si s ním umět poradit. A můžete si být jisti, že do takové situace se při tvorbě programu dostanete častěji, než vám bude zpočátku milé.

3. Použijte Experiment č.2 jako vzor pro zjištění, k jakým změnám došlo v diodách při těchto údajích:

	Předchozí stav	Aktuální stav
a.	84H	46H
b.	27H	63H
c.	02H	07H
d.	A7H	D8H

Odpovědi

1.

a.	01001010	e.	00001000 (=08H)
b.	11111111	f.	00100110 (=26H)
c.	00000000	g.	00110011 (=33H)
d.	00001110	h.	00001001 (=09H)

2. Svítit budou tyto diody LED:

a. 6, 4, 2, 0
b. 6
c. 6,5,2
d. 5
e. 1
f. 5,4
g. 2,1
h. 7,6
i. 0
j. 5,3

3. a Napřed převedeme HD čísla na binární:

84H = 10000100 předchozí stav
46H = 01000110 aktuální stav

Pak provedeme následující operace:

10000100 XOR 01000110 =11000010
10000100 AND 11000010 =10000000
NOT 10000000 = 01111111
01111111 AND 11000010 = 01000010

Svítily diody 7 a 3, teď svítí diody 6, 3 a 2. Zhasla dioda 7, rozsvítila se dioda 6. Podobně postupujte ve zbylých třech částech 3. bodu s těmito výsledky (čísla zhasnuvších a rozsvítivších se diod):

3. b	on — off 3, 2;	off — on 6
3. c	on — off žádné;	off — on 2, 0
3. d	on — off 5, 2;	off — on 6, 4, 3

KAPITOLA 5
BITOVÉ MANIPULACE, ROTACE A POSUVY

Instrukce BIT, SET a RES

Tyto instrukce zabírají značnou část instrukčního souboru Z80 - je jich celkem 240, 80 variant pro každou z nich. S jejich pomocí je nám dostupný každý bit registrů A, B, C, D, E, H, L a jakéhokoli místa hlavní paměti. Funkce instrukcí jsou tyto:

BIT – test stavu bitu (je ve stavu log.1 nebo log.0?)
SET – uvedení bitu do stavu log.1 (resp. ponechání jej v log.1)
RES – vynulování bitu (resp. ponechání jej ve stavu log.0)

Např. po provedení instrukce SET 1,B bude bit 1 reg. B ve stavu log.1. Po instrukci RES (z angl. *REStet) ve* tvaru RES 3, (HL) bude bit 3 obsahu adresy (HL) ve stavu log.0. Stav ostatních bitů na operaci zúčastněného registru nebo místa paměti zůstává nedotčen.

Instrukcí BIT 0,A testujeme log. stav bitu 0 akumulátoru. Podle toho, v jakém log. stavu testovaný bit je, bude ovlivněn indikátor Z reg. F. Je-li bit nulový, bude Z = 1, je-li ve stavu log.1, bude Z = 0. Tato instrukce nemění obsah testovaného bitu, tedy ani bajtu.

Instrukce BIT je užitečná tehdy, chceme-li testovat stav jednoho bitu bajtu. Můžeme se tak vyhnout binárnímu výpočtu masky logické operace, která navíc mění obsah maskovaného bajtu. Srovnejte:

```
1)  LD A, (1234H)        2)  LD A, (1234H)
    AND 10H                  BIT 4, A
```

V prvním případě je pro test bitu 4 obsahu adresy 1234H použit maskovací bajt 10H. Jak víme, touto operaci se všechny ostatní bity reg. A vynulují. Instrukce BIT 4,A ponechává akumulátor v původním stavu. Oba způsoby testování zabírají 5 bajtů paměti. Který z obou postupů použijeme, závisí na konkrétní programové situaci.

SKUPINA INSTRUKCÍ ROTACE A POSUVU

Těchto instrukcí je 74. Kromě čtyř z nich (RLCA, RRCA, RLA a RRA), ovlivňujících jen indikátor CY, má všech ostatních 70 vliv na všechny čtyři přístupné SI. Zvláštním typem této skupiny instrukcí jsou dvě (RLD a RRD), které mj. mohou pracovat s binárně kódovanými dekadickými čísly. Při studiu těchto instrukci i při jejich prvních programových aplikacích mějte po ruce přílohu s grafickým znázorněním jejich funkcí

ROTAČNÍ INSTRUKCE

Rotace vlevo cirkulační - RLC

Cirkulační se některé rotace jmenují proto, že v bajtu dochází k rotaci samotných osmi bitů jeho obsahu, aniž do něj vstupuje indikátor CY (jak je tomu u jiných rotací -viz dále).

Funkce RLC je patrná z obrázku přílohy. V dalším budeme značit bity mezinárodni zkratkou D (*Digit*- česky číslice) s připojeným pořadovým číslem bitu. D3 je tedy bit 3. U každé instrukce si uvedeme tabulkový příklad jejího provedení.

Instrukce RLC A provádí cirkulační rotací vlevo s bity reg. A:

	Bit CY	Akumulátor							
Před RLC A	CY	D7	D6	D5	D4	D3	D2	D1	D0
Po RLC A	D7	D6	D5	D4	D3	D2	D1	D0	D7

	Bit CY	Akumulátor							
Před RLC A	-	1	1	0	1	1	1	0	0
Po RLC A	1	1	0	1	1	1	0	1	0

Vidíme, že po každém provedení instrukce RLC A se všechny bity reg. A posunou o jeden bít doleva, přičemž CY bude nastaven na počáteční log. stav bitu 7. Pomlčka „-" na místě CY před provedením instrukce znamená, že pro instrukci samu nemá jeho předchozí stav žádný význam.

Rotace vpravo cirkulační - RRC r

Pro příklad použijeme instrukci RRC C, která provede uvedenou bitovou rotaci reg. C:

	Bit CY	Registr C							
Před RRC C	CY	D7	D6	D5	D4	D3	D2	D1	D0
Po RRC C	D0	D0	D7	D6	D5	D4	D3	D2	D1

	Bit CY	Registr C							
Před RRC C		1	1	0	1	1	1	0	0
Po RRC C	0	0	1	1	0	1	1	1	0

Instrukce RRC r je analogická instrukci RLC r. Jen je třeba pamatovat, že vlivem posunu bitů doprava se v CY objeví log. stav bitu 0.

Rotace vlevo - RL r

Provedení instrukce RL A:

	Bit CY	Akumulátor							
Před RL A	CY	D7	D6	05	D4	D3	D2	D1	D0
Po RLA	D7	D6	D5	D4	D3	D2	D1	D0	CY

	Bit CY	Akumulátor							
Před RL A	0	1	1	0	1	1	1	0	0
Po RL A	1	1	0	1	1	1	0	0	0

Zde můžeme indikátor CY zvažovat jako 9. bit (bit 8) akumulátoru. Lidově řečeno - bitová rotace probíhá „skrze" CY. To umožňuje přenos bitu CY do bajtu. Význam a užití této možnosti si ukážeme později.

Rotace vpravo - RR r

Provedení instrukce RR D:

	Bit CY	Registr D							
Před RR D	CY	D7	D6	D5	D4	D3	D2	D1	D0
Po RR D	D0	CY	D7	D6	D5	04	D3	D2	D1

	Bit CY	Registr D							
Před RR D	0	1	1	0	1	1	1	0	0
Po RR D	0	0	1	1	0	1	1	1	0

Opět jde o analogii instrukce RL r. Do CY bude přenesen bit 0. Tyto čtyři rotační instrukce mají široké využiti. Obsah rotovaných registrů může plnit funkci čítače osmi průběhů, které se tak často v programech objevují. Při rotacích se využívá testu stavu indikátoru CY. Podle sestavy log.1 a log.0 rotovaného bajtu pak lze řídit chod nějakých operací vdaném sledu sestavy. Naplňováním bajtu log stavem CY, který indikuje nějaký dvoustavový průběh, jej pak můžeme zpětně zjistit apod.

Uvedeme si příklady použití rotace. Rutina zjišťuje, zda jsou na vyšších bitových pozicích bajtu nuly a kolik jich je.

```
        LD C, N         ;Do reg. C testovaný bajt N
        LD A, 00        ;Čítač nulových bitů testovan. bajtu
KOLIK:  RL C            ;Rotace (nejvyšší bit do CY)
        JR C, KONEC     ;Když CY = 1, skok na adr. KONEC
        INC A           ;Zvýšení čítače o 1
        CP 08           ;Už rotovalo 8 bitů?
        JR NZ, KOLIK    ;Když NE, pak na adr. KOLIK
KONEC:  RET             ;Návrat
```

V případě, že na nejvyšších bitových pozicích reg. C jsou tři nuly, provedou se tři rotace se třemi zvýšeními obsahu reg. A, kde nakonec bude číslo 3. Při čtvrtém vstupu do smyčky KOLIK se do CY přesune první bit ve stavu log.1 a podmíněný skok JR C, KONEC program ukončí. Hledaný počet nul bude v reg. A.

V následujícím příkladu si ukážeme, jak můžeme inicializací bajtu stanovit, které z osmi subrutin se mají provést. Počet kombinací provedení osmi subrutin je 256:

```
RRA
CALL C, SBR1
RRA
CALL C, SBR2
RRA
CALL C, SBR3
RRA
CALL C, SBR4
RRA
CALL C, SBR5
RRA
CALL C, SBR6
RRA
CALL C, SBR7
RRA
CALL C, SBR8
```

Kombinací jedniček a nul v akumulátoru dosáhneme požadované kombinace provedení subrutin volaných instrukcemi CALL vždy, když po rotaci bude CY= 1. Pochopitelně, že v SBR1-SBR8 musí být zajištěna nedotknutelnost obsahu akumulátoru (např. PUSH AF na jejich vstupu a POP AF na výstupu). V některých případech by se mohlo stát nevýhodným to, že v průběhu jednoho průchodu sledem testů RRA nikdy nebude subrutina s vyšším pořadovým číslem provedena před subrutinou s číslem nižším.

Jistě jste si všimli, že v programu je užita instrukce RRA, zatímco vy jste se zatím seznámili s instrukcí RR A. To je určitá zvláštnost Z80. Několik instrukcí jeho souboru je funkčně takřka ekvivalentních. Liší se jen počtem bajtů, časem provedení a počtem

ovlivňovaných SI (a samozřejmě svým zápisem). Tak např. zatímco provedení dvoubajtové RR A (ta má vliv na všechny SI) trvá 8 taktů, stejnou operaci provede jednobajtové RRA za pouhé 4 takty. A to už je nějaký rozdíl! RRA však ovlivňuje jen CY. Ale o ten nám v našem programu jde, takže pochopitelně dáme instrukci RRA přednost. Podobně je tomu u instrukcí RL A a RLA. Toto zdvojení přináší instrukční kompatibilitu s mikroprocesorem 8080.

V této souvislosti vás ještě upozorním na dvě naprosto ekvivalentní instrukce, které nemají vliv ani na jeden SI. Liší se jen časem a prostorem. Jde o instrukci LD HL,(ADDR) ve dvou HD provedeních - ED6BXXXX a 2AXXXX. Samozřejmě budeme volit kratší tvar, který má i o pětinu rychlejší dobu provedení. Pokud budete programovat s pomocí generátoru strojového kódu, většinou vám ani nic jiného nezbyde, protože jejich tvůrci do nich delší tvar instrukce nezařazují. To se týká i nestandardních instrukcí, které se tak (pokud s typem vašeho mikroprocesoru fungují) musejí zapisovat přímo do paměti. Ale i bez této „extrakávy" lze programovat, co hrdlo ráčí.

INSTRUKCE POSUVU

Rozdíl mezi rotací a posuvem je v tom, že rotace probíhá v pomyslném kruhu, kdežto posuv po lince (byť někdy „zakroucené"). Pokud se vám z rotací nezatočila hlava, pojďme posouvat:

Posuv vlevo aritmetický - SLA r

Funkce SLA je patrná z obrázku přílohy. Doplníme si ji opět tabulkou:

	Bit CY	Akumulátor							
Před SLA r	-	D7	D6	D5	D4	D3	D2	D1	D0
Po SLA r	D7	D6	D5	D4	D3	D2	D1	D0	0

Příklady průběhů instrukce SLA A s různými obsahy reg. A:

	Bit CY	Akumulátor								
Před SLA A	-	0	0	1	1	0	0	1	1	= 51 dekad.
Po SLA A	0	0	1	1	0	0	1	1	0	=102 dekad.

	Bit CY	Akumulátor								
Před SLA A	-	0	1	1	0	0	1	1	0	=102 dekad.
Po SLA A	0	1	1	0	0	1	1	0	0	=204 dekad.

	Bit CY	Akumulátor								
Před SLA A	-	1	0	0	1	1	0	0	0	=152 dekad.
Po SLA A	1	0	0	1	1	0	0	0	0	= 48 dekad.

Podívejte se pozorně, co je vlastně výsledkem funkce SLA. Je to přece násobeni dvěma. Pochopitelně jen tehdy, když nedojde k přeplnění registru - to se stalo v posledním případě, proto i výsledek není očekávaných 304, protože to je číslo větší než 255. Výsledek je 2*152-256, tedy 48. Posouvat můžeme i obsahy více bajtů pomyslně umístěných vedle sebe. Například:

```
SLA E
RL D
```

je posuv obsahu párového registru DE. Podobné operace můžeme provádět i s obsahy adres:

```
LD IX, 1234H
SLA (IX)
RL (IX + 01)
```

Multibajtové posuvy bitů jsou velmi užitečné při násobení a dělení větších čísel Velký význam mají i při operacích s grafikou obrazové paměti počítačů.

Posuv vpravo aritmetický - SRA r

	Bit CY	Registr	
Před SRA r	_	D7 D6 D5 D4 D3 D2 D1 D0	
Po SRA r	D0	D7 D7 D6 D5 D4 D3 D2 D1	
	Bit CY	**Akumulátor**	
Před SRA A	_	0 0 0 0 1 1 1 1	15 dekad
Po SRA A	1	0 0 0 0 0 1 1 1	7 dekad.
	Bit CY	**Akumulátor**	
Před SRA A	-	1 0 0 0 1 1 1 0	= -114 dekad.
Po SRA A	0	1 1 0 0 0 1 1 1	= - 57 dekad

Je zde určitá analogie s funkcí SLA až na to, že místo předpokládaného naplnění bitu 7 nulou zůstává na jeho místě původní log. stav. Na první pohled vidíme, že pomocí SRA můžeme provádět aritmetické dělení dvěma. Při opakování funkce pak samozřejmě čtyřmi, osmi, šestnácti, atd. Zůstatek dělení (je-li nějaký), se objeví v indikátoru CY. Ovšem pozor - jak je patrno z tabulky, jde o dělení DK čísel (tedy vlastně 7-bitových, kde bit 7 hraje roli znaménka + nebo -)! Instrukci typu SRA můžeme použít ve spojení s RR pro vícebajtový posuv vpravo; např.:

```
SRA H
RR L
```

Posuv vpravo logický - SRL r

	Bit CY	Registr								
Před SRL r	-	D7	06	D5	D4	D3	D2	D1	D0	
Po SRL r	D0	0	D7	D6	D5	D4	D3	D2	D1	
	Bit CY	Akumulátor								
Před SRL A	-	1	0	0	0	0	0	1	1	= 131dekad.
Po SRL A	1	0	1	0	0	0	0	0	1	= 65 dekad.

Funkce SRL simuluje dělení 8-bitových čísel dvěma; zůstatek dělení se zaznamená v CY. Jak posuneme vpravo obsahy např. osmi bajtů umístěných na sousedících adresách A100H-A107H, ukazuje následující krátká rutina:

```
        LD B, 08
        LD HL, A107H
        SRL (HL)
ROT:    DEC HL
        DEC B
        JP Z, KON
        RR (HL)
        JP ROT
KON:    RET
```

Dále se budeme zabývat dvěma velmi speciálními funkcemi rotací celých polovin bajtu (skupin čtyř bitů). Poslouží nám výhodně pro rotaci celých dekadických číslic. Proto byly nazvány *Rotate Digit* (rotuj číslici):

Rotace vlevo číselná - RLD

Protože jde o rotace mezi dvěma bajty, a to akumulátoru a místem paměti určeným obsahem reg. HL, ponecháme označení bitů pro akumulátor Dx a bity paměti si označíme Bx.

	Akumulátor								Místo paměti (HL)							
Před RLD	D7	D6	D5	D4	D3	D2	D1	D0	B7	B6	B5	B4	B3	B2	B1	B0
Po RLD	D7	D6	D5	D4	B7	B6	B5	B4	B3	B2	B1	B0	D3	D2	D1	D0

A tak např.:

	Akumulátor								Místo paměti (HL)							
Před RLD	1	1	0	1	1	0	0	0	0	0	1	0	1	1	0	0
Po RLD	1	1	0	1	0	0	1	0	1	1	0	0	1	0	0	0

Analogicky rotuje skupinami čtyř sousedících bitů dvou bajtů instrukce:

Rotace vpravo číselná - RRD

	Akumulátor								Místo paměti (HL)							
Před RRD	D7	D6	D5	D4	D3	D2	D1	D0	B7	B6	B5	B4	B3	B2	B1	B0
Po RRD	D7	D6	D5	D4	B3	B2	B1	B0	D3	D2	D1	D0	B7	B6	B5	B4

A tak např.:

	Akumulátor								Místo paměti (HL)							
Před RRD	1	1	0	1	0	0	0	1	1	0	1	1	1	1	1	0
Po RRD	1	1	0	1	1	1	1	0	0	0	0	1	1	0	1	1

Všimněte si, že operací se neúčastní vyšší „půlka" reg. A. RLD a RRD se používají zvláště při tzv. BCD reprezentaci čísel. BCD *(Binary Coded Decimal)* je binárně kódované dekadické číslo. Jeden bajt může obsahovat dvě dekadické číslice, každou z nich binárně kódovanou čtyřmi bity bajtu. Např. BCD reprezentace čísel 83 a 70 je:

1 0 0 0	0 0 1 1	0 1 1 1	0 0 0 0
8	3	7	0

Dále si ukážeme vícebajtovou rotaci BCD čísel mezi adresami A100H-A103H s vynulováním nižší poloviny 1. bajtu celého řetězce:

```
        LD B, 04          ;Čítač čtyř rotací
        LD HL, A100H
        XOR A             ;Vynulování reg. A
ZNOVA:  RLD
        INC HL            ;Na další adresu
        DEC B             ;Už proběhlo 8 rotací?
        JP NZ, ZNOVA      ;Když ne, skok na adr. ZNOVA
        RET
```

	reg. A		A103		A102		A101		A100		
Počáteční stav	0	0	8	7	6	5	4	3	2	1	(poř. č.
Konečný stav	0	8	7	6	5	4	3	2	1	0	„půlek")

V následujících experimentech si ukážeme příklady použití instrukci probraných v této kapitole.

EXPERIMENT č.1

Předpokládám, že jste se již dávno seznámili s tím, co je to ASCII kód znaku. Pokud ne, podívejte se do soupisu znaků v manuálu vašeho počítače. Pro dekadické číslice 0-9 je hodnota ASCII kódů 30H-39H (tak např. číslici 5 přísluší ASCII reprezentace 35H). Kódovaných reprezentací jsme v našem učebním textu měli už celou řadu - ASCII kód je jen další v řadě. Důležité je, že byl uznán jako mezinárodní standard pro kódy nejužívanějších znaků. Proto není třeba pro každou tiskárnu kódy předefinovávat. Jinými slovy - kdo by vyrobil tiskárnu, která by se neopírala o standard ASCII, přišel by rychle na buben".

V tomto experimentu si pro procvičení převodu binárního kódu na ASCII provedeme jen jednoduchý převod binárních číslic 1 a 0 na jejich ASCII ekvivalenty 31H a 30H. Obsah reg. B - třeba 01001101 - bude převeden na 8 bajtů v ASCII ekvivalentu 30 31 30 30 31 31 30 31. Tato HD čísla jsou do paměti ukládána v opačném pořadí.

```
A100  0E09          LD C, 08          ;Čítač osmi bitů
A102  2100A2        LD HL, A200H      ;1. adresa pro ukládání ASCII kódů
A105  3630  DALSI:  LD (HL),30H       ;Uložení ASCII 30H (nula)
A107  CB40          BIT 0, B          ;Test bitu 0 reg. B
A109  2801          JR Z, NULA        ;Když bit 0 = 0, skok na adr. NULA
A10B  34            INC (HL)          ;Bit 0 = 1, zvyš 30 na 31 na adr. (HL)
A10C  23    NULA:   INC HL            ;Další adr. uložení ASCII
A10D  CB18          RR B              ;Rotace pro test dalšího bitu
A10F  0D            DEC C             ;Čítač sniž o 1
A110  20F3          JR NZ, DALSI      ;Už jich bylo 8? Když ne, pro DALSI
A112  C9            RET               ;Návrat
```

KROK 1: Před spuštěním programu inicializujte obsah reg. B dle libosti. Pak se podívejte na adresy A200H-A207H, kde budou uloženy ASCII ekvivalenty binárního obsahu reg. B. Určitou programátorskou finesou je předběžné uložení 30H (ASCII reprezentace nuly) na adresu (HL). Teprve potom se testuje, zda bit 0 reg. B je ve stavu log.0 nebo 1. Jeli 0, obsah (HL) se nezmění, je-li 1, instrukce INC (HL) zvýší 30H o 1, tedy na správných 31H.

KROK 2: Nyní si ukážeme, že i tak jednoduché zadání lze řešit více způsoby. Předchozí rutina je z rodu těch „prozaických", tedy velmi názorných. Lepší, i když pro začátečníka trochu „zakuklenější" řešení přináší následující rutina, která je asi o pětinu kratší i rychlejší. Protože jste pro její analýzu vybaveni již všemi potřebnými znalostmi, pokuste se ji pochopit bez nápovědi:

```
        LD HL, A200H
        SCF
INIC:   LD A, 18H
        RL B
        RET Z
```

```
        RLA
        LD (HL), A
        INC HL
        JR INIC
```

KROK 3: Zkuste sami upravit program tak, aby *prováděl* opačný převod (z reprezentace ASCII na binární), a porovnejte si svůj nový program s následujícím experimentem.

EXPERIMENT č. 2

Převod ASCII kódů 30H a 31H na binární tvar (0 a 1). Tedy opačný postup, než jaký byl užit v experimentu č.1. ASCII kódy jsou na adresách A200H-A207H, binární ekvivalenty jsou převáděny do reg. B.

```
A120  0E08          LD C, 08          ;Čítač osmi
A122  2100A2        LD HL, A200H      ;1.adr. ASCII kódů
A125  7E     DALSI: LD A, (HL)        ;Přenos ASCII kódu do reg. A
A126  1F            RRA               ;Bit 0 do CY
A127  CB18          RR B              ;CY do reg. B
A129  23            INC HL            ;Další adresa s ASCII kódem
A12A  0D            DEC C             ;Čítač o 1 dolů
A12B  20F8          JR NZ, DALSI      ;Už bylo 8 přenosů? Ne-li, na DALSI
A12D  C9            RET               ;Návrat
```

KROK 1: Před spuštěním programu umístěte na adresy A200H-A207H ASCII kódy 30H a 31H v libovolné kombinaci. Zde je pomocí rotací zjišťován log. stav bitu 0 obsahu adres (HL). Je-li obsahem (HL) 30H, je bit 0 = 0, je-li to 31H, je bit 0= 1. Přenos této informace do reg. B probíhá rotací přes indikátor CY.

KROK 2: Zkuste přepsat program tak, aby odebíral kódy nikoli od adresy A200H nahoru, ale od A207H dolů. Jak to ovlivní instrukce rotace?

EXPERIMENT č. 3

Ukážeme si převod BCD na ASCII reprezentaci. Každá skupina čtyř bitů BCD čísla bude převedena na jeden bajt ASCII kódu. Např. tyto 4 bajty obsahující 8 BCD číslic

0 0 1 1	1 0 0 1 = 39	(BCD)
0 1 0 1	1 0 0 0 = 58	(BCD)
0 0 0 0	0 0 0 1 = 01	(BCD)
0 1 1 1	0 0 0 0 = 70	(BCD)

budou převedeny na 8 bajtů HD ASCII kódu: 33 39 35 38 30 31 37 30. 33H je ekvivalentem BCD čísla 3, 39H BCD čísla 9 atd. Nebudeme tedy převádět celá čísla BCD každého bajtu, ale každou číslici (po dvou z každého bajtu) samostatně. Tomuto pojímání obsahu bajtů s BCD čísly se anglicky říká *„packed BCD"* (česky přibližně soubor nebo blok BCD číslic). Tyto termíny však mají poněkud jiný významový odstín, proto se i v češtině zatím používá anglický výraz.

Program převádí řetězec packed BCD bajtů z adres (HL) na ASCII kódy a ukládá je na adresy (DE). V reg. C je počet převáděných BCD bajtů (zde jsou to 4 bajty):

```
A130  3E30          LD A, 30H       ;Vyšší 4 BCD bity v reg. A budou 03H
A132  211002        LD HL, A210H    ;1. adr. bajtu „packed" BCD čísla
A135  110103        LD DE, A301H    ;1. adr. pro uložení ASCII kódu
A138  0E04          LD C, 04        ;Čítač bajtů „packed" BCD bajtů
A13A  ED67    BCD:  RRD             ;Nižší 4 bity do akum., vyšší 4
                                    ;bity (číslo 03H) už v reg. A jsou
A13C  12            LD (DE), A      ;Uložení ASCII bajtu z nižší půlky
A13D  1B            DEC DE          ;(DE) pro druhou půlku „packed" BCD
A13E  ED67          RRD             ;Vyšší 4 bity do nižší půlky akum.
A140  12            LD (DE), A      ;Uložení ASCII bajtu z vyšší půlky
A141  ED67          RRD             ;Uvedení bajtu BCD do původ. tvaru
A143  13            INC DE          ;Nastavení DE na další převod
A144  13            INC DE
A145  13            INC DE
A146  23            INC HL          ;Nastaveni HL na další převod
A147  0D            DEC C           ;Všechny BCD bajty převedeny?
A148  20F0          JR NZ, BCD      ;Když NE, skok na adr. BCD
A14A  C9            RET             ;Návrat
```

KROK 1: Program si projděte krokováním, abyste mu dobře porozuměli.

KROK 2: ASCII kódy budou uloženy na adresách (DE) A300H-A307H. Reg. DE inicializujeme vždy na adresu o 1 vyšší, protože napřed zpracováváme vyšší 4 bity bajtu BCD. Jde tedy jen o zachování dohodnutého pořadí zpracování BCD číslic. Bajty BCD jsou uloženy na adresách (HL) A210H-A213H. Znázorníme si to tabulkami:

Adresa	Vyšší 4 bity	Nižší 4 bity	
A210	BCD1	BCD2	
A211	BCD3	BCD4	
A212	BCD5	BCD6	ASCII kód
A213	BCD7	BCD8	sestaven po

A300	ASCII1	2. RRD
A301	ASCII2	1. RRD
A302	ASCII3	5. RRD
A303	ASCII4	4. RRD
A304	ASCII5	8. RRD
A305	ASCII6	7. RRD
A306	ASCII7	11. RRD
A307	ASCII8	10. RRD

	Akumulátor		Adr. 0210H	
Po inicializaci reg. A	3	0	BCD1	BCD2
Po 1. RRD (ASCII2 na 0301)	3	BCD2	0	BCD1
Po 2. RRD (ASCII1 na 0300)	3	BCD1	BCD2	0
Po 3. RRD	3	0	BCD1	BCD2

Po 3. RRD je BCD bajt uveden do původního stavu a reg. A opět inicializován do vstupního stavu. Protože po uložení ASCII1 je v reg. DE A300H, a my budeme ukládat ASCII4 na adr. A303H, musíme obsah reg. DE zvýšit o 3 (třikrát INC DE).

Ptáte-li se, proč program tak složitě nakládá se skupinami 4 bitů, když by bylo možno převést BCD2 na adresu A300H, BCD1 na A301H atd., je to proto, že kdybychom nedodrželi uvedené pořadí, čísla by se vytiskla např. na tiskárně pozpátku, zpřeházeně. Pokud bychom se však přece jen dali tou kratší cestou, museli bychom nějakou pomocnou rutinou pořadí jejich uložení (nebo odeslání na výstup pro tisk) přehodit dodatečně

Poslední experimenty vám poodkryly roušku zahalující tajemství uplatnění strojně kódových programů v praxi výpočetní techniky. Čísla převedená uvedeným způsobem do ASCII kódu mohou být přímo odeslána do rutin zobrazení na monitoru nebo do tiskové rutiny interfaců připojeného k tiskárně. Způsobů programového zpracováni čísel a znaků je nepřeberné množství. Vždy jde o jejich kódování a dekódování. Jednou ze zvláštních forem práce s daty je jejich komprimace. Výrazně šetří paměť počítače hlavně v případě užití databank. Slučuje skupiny písmen, která se tak do paměti nezapisují jednotlivě, ale jako jeden kód. Komprimace je vědou nabízející řadu variant pro práci s větším počtem dat. Nakonec však vždy před svým odesláním na tiskárnu nebo obrazovku musejí být převedena na ASCII kód, abychom si navzájem rozuměli.

KAPITOLA 6

ARITMETICKÉ INSTRUKCE A BLOKOVÉ PROHLEDÁVÁNÍ

OSMIBITOVÉ ARITMETICKÉ INSTRUKCE

Provádí se jimi sčítání (ADD, ADC) a odečítání (SUB, SBC) bajtů. K této skupině instrukcí patři i INC (zvýšení obsahu bajtu o 1) a DEC (jeho sníženi o 1). Všechny tyto instrukce ovlivňují všechny indikátory. Až na jednu výjimku — INC a DEC neovlivňují stav indikátoru CY. Na to pozor, v tom se často chybuje! Zapomětliví programátoři testují přeplnění obsahu bajtu, na které u těchto dvou instrukcí nemohou být nikdy upozorněni. Chybu pak většinou hledají tam, kde není. Skupinu uvedených instrukcí doplňují ještě instrukce CP a speciální DAA. Instrukcemi ADD (SUB) přičítáme k (odečítáme od) akumulátoru obsah druhého bajtu operace se účastnícího. Např. ADD A, B znamená, že k obsahu reg. A přičteme obsah reg. B. Při SUB (HL) od obsahu reg. A odečítáme obsah adresy (HL). Apod.

Indikátory S, Z, P/V a CY se chovají jak náleží. Opět nezapomeňte, že v případě P/V jde o test přeplnění DK čísel. I na to se zpočátku často zapomíná.

I když indikátory H a N přímo testovat nemůžeme, je jejich stav v tabulkách uveden, protože nám někdy při krokování programem může log. stav H říci, zda došlo či nedošlo k přenosu mezi bity 3 a 4 uvnitř bajtu - pak je H ve stavu log. 1. To je užitečné především při práci s BCD čísly. Indikátor N je ve stavu log.1 po každé operaci odečítáni.

	HD obsah akumulátoru		Indikátory po provedeni					
Instrukce	před	po	S	Z	H	P/V	N	CY
INC A	04	05	0	0	0	0	0	.
INC A	FF	00	0	1	1	0	0	.
DEC A	00	FF	1	0	1	0	1	.
ADD A, 80H	00	80	1	0	0	0	0	0
ADD A, 70H	80	F0	1	0	0	0	0	0
ADD A, F0H	F0	E0	1	0	0	0	0	1
ADD A, 11H	22	33	0	0	0	0	0	0
ADD A, 18H	29	41	0	0	1	0	0	0
ADD A, 94H	93	27	0	0	0	1	0	1
ADD A, 99H	99	32	0	0	1	1	0	1
SUB 33H	33	00	0	1	0	0	1	0
SUB 02H	10	0E	0	0	1	0	1	0
SUB 22H	10	EE	1	0	1	0	1	1

Tečka u CY znamená, že se jeho předchozí stav nemění, instrukce jej neovlivňuje.

Pokud jste novopečenými adepty programování, pro pochopení průběhu uvedených instrukcí musíte mít buď vysokou představivost, nebo (lépe) tužku a papír, abyste si mohli zapsat vstupní i výstupní binární kódy operací jednotlivých instrukcí. Proberme si třeba instrukci ADD A,18H s počátečním obsahem reg. A 29H:

```
  0 0 1 0 1 0 0 1
+ 0 0 0 1 1 0 0 0
-----------------
  0 1 0 0 0 0 0 1
```

Stavy indikátorů:

S = 0 (bit 7 = 0)
Z = 0 (výsledek je nenulový)
H = 1 (došlo k přenosu z bitu 3 do bitu 4)
P/V = 0 (součet dvou DK kladných čísel je opět kladné DK číslo)
N = 0 (operace není odečítáním)
CY = 0 (nedošlo k přeplnění bajtu, resp k přenosu z bitu 7)

Obdobou instrukcí ADD a SUB jsou ADC a SBC. Změna je pouze v jediném — k výsledku součtu (resp. od výsledku odečtu) se přičte (resp. odečte) log. stav indikátoru CY. Při CY = 0 bude výsledek stejný jako při užití instrukcí ADD a SUB. Bude-li log. stav CY=1, pak při ADC se k výsledku součtu ještě přičte číslo 1, při SBC se 1 odečte. Záleží na log. stavu CY těsně před provedením instrukce. Příklady:

	reg. A	CY	reg. A	Indikátory po provedení					
Instrukce	před	před	po	S	Z	H	P/V	N	CY
ADC A, 00H	01	1	02	0	0	0	0	0	0
ADC A, 00H	01	0	01	0	0	0	0	0	0
ADC A, 90H	97	1	28	0	0	0	1	0	1
ADC A, 19H	39	1	53	0	0	1	0	0	0
SBC A, 00H	00	1	FF	1	0	1	0	1	1
SBC A, 01H	02	1	00	0	1	0	0	1	0
SBC A, 80H	00	1	7F	0	0	1	0	1	1

Instrukce ADC a SBC jsou velmi užitečné při sečítání a odečítání vícebajtových čísel právě díky účasti indikátoru CY, který nám pomáhá při přenosu bitu ze „sousedícího" bajtu v jejich řadě. Vzdáleně to připomíná účast CY při vícebajtovém násobení a dělení instrukcemi rotací a posuvů. Vzpomínáte? Jestli ne, zřejmě učení nevěnujete potřebné úsilí. Při programování pak budete knížkou listovat víc, než by bylo zdrávo, a nakonec se může stát, že to vzdáte. Strojový kód chce klid a koncentraci. Jeho logika je precizní, i když zpočátku těžko stravitelná. Ale jakmile ji jednou pochopíte, všechno půjde takřka samo. O to víc se budete moci věnovat tvůrčímu myšlení, než abyste ztráceli čas při hledání ve stozích papírů ..jakže se to tady, ne, to není ono, kde jen to ... atd. Jediné,

co byste měli mít při programování po ruce, je soupis tvarů zápisu jednotlivých instrukcí a jejich vlivu na SI (při precizním časování rutin i prováděcí časy instrukcí). A samozřejmě poznámkový papír pro tvorbu struktury programu.

Nyní si ukážeme sečítání dvou sčítanců, z nichž každý obsahuje 8 bajtů:

```
        LD C, 08          ;Počet bajtů v každém ze sčítanců
        LD HL,A200H       ;Adresa 1. bajtu prvního sčítance
        LD IX,A210H       ;Adresa 1. bajtu druhého sčítance
        LD IY, A220H      ;1. adresa uložení výsledku
        XOR A             ;CY = 0 (pro 1. součet)
ADDB:   LD A,  (HL)       ;1. bajt 1. sčítance
        ADC A, (IX+ 00)   ;Přičtení 1. bajtu 2. sčítance
        LD (IY + 00), A   ;Uložení výsledku na adr. (IY)
        INC HL            ;Zvýšení všech adres pro další
        INC IX            ;součet
        INC IY
        DEC C             ;Už bylo provedeno 8 součtů?
        JR NZ, ADDB       ;Když NE, skok na adr. ADDB
        JR C, CARRY       ;Pokud CY = 1 po 8. součtu, (viz text)
        RET               ;Návrat
```

Funkce programu je naprosto průzračná. Po vstupních inicializacích reg. C, HL, IX, IY a A instrukce XOR A zařídí, aby indikátor CY byl vynulován a nepřešel jako 1 do prvního součtu. Pro vynulováni CY bychom také mohli použít AND A, OR A či sled instrukcí SCF a CCF.

Pokud dojde k přeplnění registru A po některém ze součtů, nastaví se CY na log. 1 a ta se pak přičte k příštímu součtu vyššího řádu. Podobně přece postupujeme při sečítání dekadických čísel. Když nám třeba stovky „přetečou", přesuneme „CY" jako jedničku do tisícovek. Analogie s výše uvedeným programem je zřejmá.

Když se objeví CY= 1 i po posledním součtu (a onu 1 už nemáme kam přičíst), skočí program na subrutinu CARRY, která tu není uvedena. Ta buď provede patřičnou „kosmetickou úpravu" výsledku, aby odpovídal pravdě, nebo nám jen ohlásí, co se stalo.

Kdybychom tento program chtěli použít pro součet DK čísel, museli bychom jej upravit pro test a přenos indikátoru P/V. Zkuste upravit sami. Ale na počítači! Na papíru se váš program může tvářit zcela nevinně, ale pak třeba ... ouha!

Instrukce DAA

Jde o speciální aritmetickou instrukci pro práci s BCD čísly. Převádí binární číslo obsažené v akumulátoru na formát BCD. Z80 zná jen jedno sečítání a odečítání - binární. Mikroprocesoru je jedno, zda sečítáme čísla binární, či DK. Pro nás se operace liší jen testem patřičného SI - CY nebo P/V. U dekadických čísel to však už tak jednoduché není. Posuďte sami na přikladu, kde je uveden postup sečítání čísel binárních i BCD:

```
00001000  =    8(dekad.)  či 08 (packed BCD)
00001001  =    9(dekad.)  či 09 (packed BCD)
--------
00010001  =   17(dekad.)  či špatně 11 (packed BCD)
```

Binární výsledek je správný. Ale reprezentace BCD je chybná. Příčina spočívá v číselném základu. Dekadické číslo má základ 10 (BCD jej musí mít taky 10). BCD je tu však mikroprocesorem zvažováno jako číslo se 16 možnými kombinacemi, tedy hexadekadické. My však chceme použít jen 10 kombinací. V uvedeném součtu došlo k přenosu z bitu 3 do bitu 4, což signalizuje, že výsledek je větší než 16. Výsledek 11 tedy v tomto případě není BCD, ale HD. Pro správnou reprezentaci BCD při aritmetických operacích by se tedy měl přenos objevit nikoli při dosažení čísla 16, ale 10. Proč věc takhle rozebírám, se dozvíte za chvíli.

A tak binární součet není BCD číslem,

1. když je součet skupiny 4 bitů v intervalu 10.. 15:

```
1001  9
0010  2
----
1011  B jako HD číslo (ale mělo by být 11 jako BCD).
```

V tomto případě nedochází k přenosu mezi bity 3 a 4 (H = 0).

2. Když je součet skupiny 4 bitů větší než 15:

```
 1001  9
 1001  9
-----
10010 12 jako HD číslo (ale mělo by být 18 jako BCD).
```

Zde dochází k přenosu mezi bity 3 a 4 (indikátor H = 1), ale pro účely BCD má „zpoždění šesti jednotek". Právě proto byla instrukce DAA vybavena schopností zjistit, kdy dojde k jedné z obou uvedených eventualit, a patřičnou šestku pak přičte k obsahu skupiny 4 bitů. Po přičtení čísla 6 pro „dohnání zpožděni" v prvním případě bude binární obsah obou skupin 4 bitů v bajtu 00010001, což je správné BCD číslo 11. Ve druhém případě to bude 00011000, což je BCD reprezentace 18.

V prvním případě, kdy se ve skupině 4 bitů objeví HD číslo A,B,C,D,E nebo F (tedy víc než 9, ale míň než 16), Z80 přičte šestku. Určitou zvláštností je, že se 6 přičte, i když se neindikuje žádná změna v reg. F. Indikace je vnitřní, mimo reflexi reg. F.

Ve druhém případě, kdy dojde k přenosu, je tento přenos signalizován indikátorem H (při součtu nižších 4 bitů) nebo CY (přeplnění bajtu při součtu vyšších 4 bitů) Z80 přičte 6 k výslednému obsahu té které půlky bajtu. V případě vyšší půlky bajtu to z hlediska dekadického můžeme pojímat jako přičtení čísla 60 k celému výsledku. V případě současného výskytu obou přenosů se vlastně přičte 66.

Funkci instrukce DAA si opět vysvětlíme tabulkově. Použijeme pro to sled dvou instrukcí - DAA převede výsledek binárního součtu reg. B a reg. A na BCD číslo:

```
ADD A, B
DAA
```

Reg. B i A budeme inicializovat na pět různých hodnot:

	1.	2	3.	4.	5.
Reg. B	11	19	91	99	09
Reg. A před ADD A, B	22	18	81	88	05
Po ADD A, B: reg. A	33	31	12	21	0E
H (+ 6)	0	1	0	1	0
CY (+ 60)	0	0	1	1	0
Reg. A po DAA	33	37	72	87	14

Ve sloupci 3. a 4. je součet vyšší než 99, což je nejvyšší BCD číslo, jaké může bajt po operaci DAA obsahovat. Jednička stovky tedy „přetekla" (CY=1) a v bajtu zůstal výsledek menší o 100. Toto přeplnění můžeme přenést zase buď do jiného bajtu, nebo s ním v programu pracovat nějak jinak - jak zrovna potřebujeme. O použití této řádové jednotky můžete přemýšlet, když si poslední uvedený program multibajtového součtu binárních čísel přeměníte na multibajtový součet BCD čísel - stačí jen vložit DAA mezi instrukce ADC A, (IX+ 00) a LD (IY + 00),A

Pro použití instrukce DAA je třeba mít na paměti, že ji musíme zařadit ihned po součtu, protože DAA potřebuje znát log. stavy SI poslední součtové operace.

16-BITOVÉ ARITMETICKÉ INSTRUKCE

Všechny předchozí instrukce - kromě SUB a DAA — mají své 16-bitové analogie. Liší se v zápisu, registrech, s nimiž operují, a v ovlivňováni SI reg. F. Zásadně si pamatujte, že všechny 16-bitové instrukce INC rr a DEC rr neovlivňují vůbec žádné indikátory. Zařazením testů SI po INC rr a DEC rr zjistíte jejich stav před provedením těchto instrukcí! Na druhou stranu je výhodou, že DEC rr ani INC rr indikátory neovlivňují. Tak můžeme snižovat či zvyšovat obsahy reg. BC, DE, HL, IX, IY, SP podle potřeby těsně před momenty, v nichž se rozhoduje o větvení programu na základě výsledků předchozích operací.

Instrukce porovnávání - CP r

Funkce CP je shodná s funkci instrukce SUB až na to, že po jejím provedení se nezmění obsah akumulátoru. Proto se říká, že tato instrukce jen porovnává. Její užití je velmi výhodné vždy, když v rozhodovacích bodech programu testujeme obsah akumulá-

toru, aniž ho chceme změnit. Instrukce ovlivňuje všechny SI v závislosti na výsledku porovnání (odečtu). Funkci instrukce CP B si můžeme namodelovat jako tento sled instrukcí.

```
LD C, A         ;Uschování obsahu reg. A do reg. C
SUB B           ;Odečet B od obsahu reg. A
LD A, C         ;Původní stav reg. A zpět do reg. A
```

Výsledný obsah akumulátoru se nezměnil, ale všechny SI indikují výsledek instrukce SUB B. V jednom z předchozích programů, v němž jsme si demonstrovali konstrukci skokové tabulky, jsme se už s funkcí CP setkali.

Instrukce prohledávání bloku dat

Jsou čtyři - CPI, CPD, CPIR a CPDR. Jsou vzdálenou analogií instrukcí přenosu bloku dat (LDI, LDD, LDIR a LDDR). Liší se hlavně v tom, že instrukce prohledávání obsahují princip funkce CP, tedy porovnáváni:

Reg. A - obsahuje bajt, jehož ekvivalent hledáme v bloku dat.
Reg. BC - obsahuje počet adres, jejichž obsah chceme porovnávat s obsahem reg A
Reg. HL - obsahuje adresu (u instrukcí s opakováním 1. adresu prohledávaného bloku), jejíž obsah (obsahy) budeme porovnávat s obsahem reg. A.

Na rozdíl od funkce CP prohledávání neovlivňuje log. stav CY! Prohledávání ovlivňuje indikátory Z, S a H.

Průběh provedení **instrukce CPI (***ComPare-Increment):*

1. Bajt na adrese (HL) je porovnán s obsahem akumulátoru. Indikátory Z,S,H jsou ovlivněny výsledkem porovnání.
2. Obsah reg. HL je zvýšen o 1 (ve smyslu INC HL).
3. Obsah reg. BC je snížen o 1 (ve smyslu DEC BC) - Indikátor Z = 1, když je obsah akumulátoru shodný s obsahem adr. (HL), tedy když výsledkem porovnání je nula. P/V = 0, když je obsah BC nulový (jinak je P/V=1).

Průběh provedení **instrukce CPIR** *(ComPare-Increment-Repeat),* čili porovnávaní s opakováním:

V bodech 1. až 3. jako u CPI.

4. Pokud obsah akumulátoru není shodný s obsahem adresy (HL), nebo obsah reg BC není nulový, instrukce se opakuje opět od bodu 1, dokud není splněna některá z obou uvedených podmínek.

Instrukce CPD i **CPDR** jsou analogické až na to, že obsah reg. HL je snižován o 1 (ve smyslu DEC HL). U CPI a CPIR jde tedy o prohledávání bloku dat směrem k vyšším adresám, u CPD a CPDR k nižším. Případně nalezený bajt leží na adrese (HL—1) u CPIR nebo (HL+1) u CPDR

EXPERIMENT č.1

Program vynásobí dvě 16-bitová binární čísla uložená na adresách A130H-A131H a A132H-A133H. Výsledek je uložen na adresy A134H-A135H. Později si ukážeme i násobení 64-bitových čísel.

```
A100  210000          LD HL, 0000       ;Inicializace vstupních hodnot
A103  ED5B30A1        LD DE, (A130H)    ;registrů
A107  ED4B32A1        LD BC, (A132H)
A10B  7A              LD A, D           ;Test- je DE = 0?
A10C  B3              OR E
A10D  C8              RET Z             ;Když ANO, návrat (výsledek = 0)
A110  CB38      NAS:  SRL B             ;Když NE, posuv a rotace - CY ob-
A112  CB19            RR C              ;sahuje hodnotu násobícího bitu
A114  3004            JR NC, NCF        ;Je CY = 1?
A116  19              ADD HL, DE        ;Ano, pak přičti DE k HL
A117  D8              RET C             ;Při přeplnění po součtu návrat
A11A  78        NCF:  LD A, B           ;CY = 0. Test- je BC = 0?
A11B  B1              OR C
A11C  CA29A1          JP Z, VYSL        ;Když ANO, konec, návrat
A11F  CB23            SLA E             ;Když NE, posuv a rotace
A121  CB12            RL D
A123  D8              RET C             ;Při CY=1 návrat
A126  C31001          JP NAS            ;Když CY = 0, pokračování násobení
A129  2234A1    VYSL: LD (A134H), HL    ;Uložení výsledku násobení
A12C  C9              RET               ;a návrat
```

KROK 1: Program si probereme trochu podrobněji, protože se v něm odehrává řada procesů, které na první pohled nemusejí být zřejmé. Princip si ukážeme zjednodušeně na násobení 4-bitovém. Lze jej však aplikovat na násobení s libovolným počtem bitů. Předpokládejme, že chceme číslo 0101 násobit 0011 krát (čili třikrát). Jedna z možných procedur provedení operace je shodná s běžným dekadickým násobením:

```
        0 0 1 1
      * 0 1 0 1
-----------------
        0 0 1 1     1*0011
      0 0 0 0       0*0011
    0 0 1 1         1*0011
  0 0 0 0           0*0011
-----------------
  0 0 0 1 1 1 1
```

Z toho je zřejmá jedna zákonitost - po každém dílčím násobku se jeho (dílčí) výsledek posune o 1 místo doleva Násobíme-li „dílčí" nulou, je pochopitelně dílčí výsledek

nulový. Ale při násobení jedničkou se na posouvanou pozici promítne obsah násobeného čísla. Tyto dílčí násobky se nakonec sečtou (nebo se mohou sečítat průběžně). Při operacích posuvu vlevo (SRL) naplňujeme registr nulami zprava, což nám plně vyhovuje V našem programu reprezentuje DE číslo násobené, BC počet dílčích násobení, do HL je ukládán výsledek násobení.

KROK 2: Zkuste projít program s různými HD obsahy reg. DE a BC, např.:

```
0400*0020=8000
00FF*00FF=FE01
0100*00FF=FF00
```

Ale co v případě: `0100*0100 = ????`

Výsledek v HL bude 0000, což je špatně. Správný výsledek je tříbajtový: 010000. Ale my do HL ukládáme dva bajty. Pro tento případ přeplnění by bylo nutno program rozšířit. Testy přeplnění jsou v programu na dvou místech (pomocí instrukcí RET C): po posuvu násobeného čísla v reg. DE a po přičtení dílčího násobku k reg. HL. Úprava není složitá. Zkuste ji vytvořit sami.

KROK 3: Předchozí techniku násobení dvoubajtových binárních čísel můžeme lehce převést na násobeni vícebajtové. Zkuste pro ně upravit předchozí program a vyzkoušejte si správnost vaší úpravy.

EXPERIMENT č. 2

Odečet N-bajtových BCD čísel. Obě čísla jsou uložena od adres XN a YN, kde leží jejich nejnižší bajty. Rozdíl je postupně ukládán od adresy ZN.

```
A200  06NN            LD B, N          ;V reg. B počet odečtu (bajtů)
A202  D021X2X1        LD IX, XN        ;Inicializace adres
A206  FD21Y2Y1        LD IY, YN
A20A  21Z2Z1          LD HL, ZN
A20D  37              SCF              ;CY=1; stovkový doplněk pro 1. odč.
A20E  3E99      DALSI: LD A, 99        ;Hledání doplňku 99 nebo 100
A210  CE00            ADC A, 00        ;menšitele (CY = 0 nebo 1)
A212  FD9600          SUB (IY + 0)     ;Odečet s CY
A215  DD8600          ADD A, (IX + 0)  ;Přičtení menšence
A218  27              DAA              ;Převod na dek. číslo
A219  77              LD (HL),A        ;Uložení výsledku
A21A  DD23            INC IX           ;Přechod na další bajty
A21C  FD23            INC IY
A21E  23              INC HL
A21F  10E8            DJNZ DALSI       ;Na DALSI bajt, dokud není B=0
A221  C9              RET              ;Návrat
```

KR0K 1: Vzpomeňte si na odečítání DK čísel - díky jejich bajtové konstrukci je DK odečítání shodné se součtem dvojkových komplementů (doplňků) každého z nich. Totéž platí pro čísla BCD - ovšem místo formování DK musíme pro součet určit komplement stovkový (jen pro tuto pasáž zkráceně SK). Pokud je vám věc komplementů jasná, přijměte kompliment. Ale pro jistotu si provedeme názorné příklady:

```
BCD bajt 03   94   30   01   50
SK       97   06   70   99   50
```

Až příliš triviální. SK k decimálnímu dvoumístnému číslu najdeme tak, že toto číslo odečteme od čísla 100. Podobně bychom mohli najít desítkový komplement pro půlky bajtů, ale to nepotřebujeme. Podívejme se teď na uplatnění techniky sečítání SK v případě odečítání tříbajtových čísel — třeba 256925 - 133639:

1. Určíme SK nejnižšího bajtu čísla odečítaného (menšitele):

   ```
   100-39 = 61
   ```

2. Výsledek přičteme k nejnižšímu bajtu čísla menšeného (menšence):

   ```
   25 + 61 =86 (nedošlo k přenosu)
   ```

3. Protože nedošlo k přenosu (výsledek není větší než 99), určíme 99-komplement dalšího bajtu menšitele:

   ```
   99-36 = 63
   ```

4. Výsledek opět přičteme k dalšímu bajtu menšence:

   ```
   69 + 63 = 32 (došlo k přenosu!)
   ```

5. Protože došlo k přenosu (tedy přenosu 1 do vyššího řádu), určíme SK k dalšímu (zde již nejvyššímu) bajtu menšitele:

   ```
   100-13 = 87
   ```

6. Výsledek přičteme k nejvyššímu bajtu menšence:

   25 + 87 = 12 (došlo k přenosu!)

7. Výsledek celého SK součtu (tedy odečtu BCD čísel) je 123286, což je správně. Správnost výsledku je potvrzena log. stavem CY = 1. Pokud by byl CY = 0, výsledek by byl chybný. Protože však BCD čísla jsou vždy větší než nula, nebo jí rovna, přenos se objeví vždy, kdykoli je menšenec větší než menšitel.

Co do přenosu je sečítání komplementů opačným procesem odečítání nekomplementámích čísel: Přenos (CY = 1) při odečítání čísel je ekvivalentní nulovému přenosu (CY = 0) při součtu jejich komplementů.

V našem experimentu není závěrečný test přeplnění indikován. Ale můžete si jej doplnit sami. Otestujte si jeho správnou funkci na počítači.

EXPERIMENT č. 3

Dělení 16-bitového čísla (v reg. HL) 8-bitovým (v reg. D). Výsledek dělení se objeví v reg. L, přičemž v reg. H bude zůstatek dělení. Pro to, aby byl výsledek (jeho celočíselná část) 8-bitový (aby se „vešel" do jednoho registru), musejí být splněny tyto podmínky:

1. 16-bitový dělenec musí mít nejvyšší bit nulový,
2. vyšší bajt dělence musí být menši než dělitel.

```
A2FB  21NNNN          LD HL, NNNN     ;Uložení dělence
A2FE  16NN            LD D, NN        ;Uložení dělitele
A300  0608            LD B, 08        ;Čitač bitů dělitele
A302  1E00            LD E, 00        ;Vynulování reg. E pro SBC HL, DE
A304  29        DALSI: ADD HL, HL     ;Posuv obsahu HL vlevo, vynulová-
                                       ;ní nejnižšího bitu
A305  AF              XOR A           ;Vynulování CY
A306  ED52            SBC HL, DE      ;Test pro děleni — je další bit
A308  23              INC HL          ;0 nebo 1?
A309  3002            JR NC, HUP      ;Je-li nula, přeskok na HUP
A30B  19              ADD HL, DE      ;Přičtení DE zpět k HL
A30C  2B              DEC HL          ;Odečtení 1 od nejnižšího bitu
A30D  10F5      HUP:  DJNZ DALSI      ;Na DALSI, dokud B není nula
A30F  C9              RET             ;Návrat
```

KROK 1: Užitý algoritmus je podobný vžitému dělení dekadickému. Je však jednodušší o to, že počítáme jen s číslicemi 1 a 0. Věc si probereme na dělení 16-bitového čísla 8EH (01101110) 8-bitovým 08H (1000). Je zde analogie s výše provedeným vysvětlením násobení. Stejně tak i při dělení „vynecháváme odečítáni", když se v dělenci objeví nulový bit:

```
01101110:1000=1101
-1000

 1011
 -1000

  1110
  -1000

00000110
```

Výsledek je 1101 (0DH) se zůstatkem 0110 (06H).

Osvětlíme si, jak uvedený postup probíhá v experimentu. Protože nejvyšší bit reg. HL je nulový (vstupní podmínka), můžeme provést jeho posuv vlevo - to je totéž, co násobení dvěma, tedy totéž, co ADD HL, HL. Přitom se vynuluje nejnižší bit reg. HL. Tímto posouváním (s následnými změnami vlivem dalších operací) se nám nakonec celočíselná část výsledku objeví v reg. L. Porovnáme obsahy HL a DE. Když je DE menší než, nebo rovno HL, výsledek porovnání (v CY) je 1, pak se provede instrukce INC HL, kterou se k nejnižšímu bitu reg. L přičte (byl předtím posuvem vynulován) hodnota 1; bude tedy ve stavu log. 1. Když je DE větší, bude výsledkem 0 (CY=0). Pak musíme do reg. HL vrátit jeho původní obsah (instrukcí ADD HL, DE) se zrušením předtím zvýšené hodnoty nejnižšího bitu reg. HL - čili INC HL se vyneguje instrukcí DEC HL. Při tom všem nezapomeňte, že obsah reg. E je stále nulový (je tedy poněkud „mimo hru", nikoli však doslova). Dělení probíhá mezi registrem H a D. Jde o naprostou analogii s výše uvedeným „papírovým" dělením. Posuvem postupně uvolňujeme nepotřebné bity reg. L, do nichž ukládáme dílčí výsledky dělení. Dílčí zůstatky jsou v reg. H. Po konečném odečtu zůstane v reg. H poslední zůstatek, v reg. L „naposouvaný" celočíselný výsledek. Instrukci SBC HL, DE používáme proto, že v instrukčním souboru Z80 neexistuje 16-bitová instrukce typu SUB. Protože SBC odečítá i hodnotu log. stavu CY, musíme jej před provedením instrukce vynulovat - zde pomocí XOR A.

Není-li vám cokoli jasné, proveďte si krokování programem.

EXPERIMENT č. 4

Užití instrukcí porovnávání a blokového prohledávání.

```
; Program 1 — hledání určeného znaku v řetězci různých znaků
;
A400  2100AA            LD HL, AA00H    ;1 adresa řetězce
A403  012000            LD BC, 0020H    ;Počet znaků v řetězci
A406  3E24              LD A, 2AH       ;Hledaný znak (ASCII kód „.")
A408  EDB1              CPIR            ;Blokové prohledání
A40A  C20F04            RET NZ          ;Když Z = 0, "*" nenalezen
A40D  2B                DEC HL          ;HL-1 = adresa se znakem "*"
A40E  03                INC BC          ;BC +1 = pořadí "*" v řetězci
A40F  C9                RET             ;Návrat
;
; Program 2 - hledání třibajtového řetězce v tabulce dat (šest
;             záznamů, z nichž každý má délku 8 bajtů)
;
A415  2100AC            LD HL, AC00H    ;Adresa posledního záznamu
A418  010600            LD BC, 0006     ;Počet záznamů v tabulce
A41B  3A00AB            LD A, (AB00H)   ;1.znak hledaného řetěz. do reg. A
A41E  11F9FF            LD DE, FFF9H    ;Dvoj.kompl.dé!ky záznamu plus 1
A421  EDA9      DALSI:  CPD             ;Porovnáni prvních bajtů
A423  2809              JR Z, POR       ;Když O.K, porovnej další
```

```
A425  E2001A          JP PO, KONEC   ;Všechny zázn. otest.? ANO-navrat
A428  19      OBNOV:  ADD HL, DE     ;NE-do HL adresa dalšího záznamu
A429  3A00AB          LD A, (AB00H)  ;Do reg. A opět 1. znak řetězce
A42C  18F3            JR DALSI       ;Skok-porov 1. bajtu dalšího zázn.
A42E  3A01AB  POR:    LD A, (AB01H)  ;Do reg. A 2. bajt řetězce
A431  E5              PUSH HL        ;V HL je poslední adr. předchozího
A432  DDE1            POP IX         ;záznamu-její přenos do reg. IX
A434  DDBE02          CP (IX+02)     ;Porovnání 2. bajtů
A437  20EF            JR NZ, OBNOV   ;Nález? Když NE,obnov. parametrů
A439  3A02AB          LD A, (AB02H)  ;Když ANO, porovnání posledních
A43C  DDBE03          CP (IX + 03)   ;bajtů
A43F  20E7            JR NZ, OBNOV   ;Nález? Když NE,obnov. parametrů
A441  23              INC HL         ;V HL 1. bajt obsahu záznamu
A442  C9      KONEC:  RET            ;Návrat
```

Oba programy jsou představiteli velmi často užívané metody prohledávání bloku dat. Jejich užití je velmi široké. Např. když potřebujeme zjistit, jaké tlačítko bylo stisknuto, a na základě toho z tabulky vyvolat nějakou akci (jde tedy o sloučení obou programů dohromady). Prohledávání bloku dat je základním principem všech databázových programů. Prakticky nenajdete program, v němž by se něco neporovnávalo s něčím pomocí některé z instrukcí porovnávání, ať bez opakování či s ním.

Program 1 demonstruje způsob zjištění, zda bajt dané hodnoty je obsažen v bloku dat. Vše je naprosto jasné, pokud si pamatujete průběh provedení instrukce CPIR (pokud ne, vraťte se k ní). Když nedojde k nálezu, Z = 0. V opačném případě byl hledaný bajt nalezen - jeho obsah je shodný s obsahem reg. A. Potřebujeme-li znát jeho adresu, musíme snížit HL o 1, pro zjištění jeho pořadí v tabulce zvýšíme BC o 1.

Program 2 ukazuje princip práce databanky při zjišťování, zda obsahuje hledaný záznam, a když ano, na jaké adrese je uložen. Předpokládejme, že máme v paměti počítače takovouto tabulku záznamů:

			Identifikace		Další element záznamu					
Adresa	Záznam č.	Bajt č:	1	2	3	4	5	6	7	8
AB08H	1		41	51	46	31	32	33	30	36
ABE0H	2		42	46	47	33	36	30	30	34
ABE8H	3		41	42	43	36	35	34	32	31
ABF0H	4		43	42	41	36	36	36	36	36
ABF8H	5		43	41	42	34	33	34	33	34
AC00H	6		42	42	42	31	32	33	32	31

Tabulka obsahuje 6 záznamů. HD čísla na jednotlivých adresách tabulky jsou ASCII kódy znaků -v tomto případě čísel a písmen. Nám jde při hledání o obsah dalšího elementu záznamu (bajty 4-8), kterým může být např. telefonní číslo osoby, jejíž šifra je obsažena v identifikačním elementu téhož záznamu (bajty 1-3). Identifikační šifru zde

reprezentuje náš řetězec tři bajtů, které postupně porovnáváme s řetězcem tří bajtů (1-3) každého záznamu. Když je hledání úspěšné, program nám obsahem reg. HL hlásí, od jaké adresy záznam (jeho 1. bajt) začíná. Tak např. hledáme-li telefonní číslo osoby s identifikační šifrou BBB (ASCII 424242), která je v záznamu 6, v HL bude adresa AC00H a z adres AC00H + 3 až AC00H + 8 si můžeme vyvolat její telefonní číslo 12321 (ASCII 3132333231). Samozřejmě, že šifru BBB musíme uložit (v ASCII kódu) na adresy AB00H-AB02H, odkud jsou jednotlivé bajty postupně přenášeny do reg. A pro porovnávání pomocí instrukce CP.

Analýza Programu 2 - HL obsahuje adresu posledního záznamu (AC00H). BC je čítačem počtu záznamů (je jich 6). DK obsah DE je nastaven tak, aby přičítal -7 k obsahu HL. Nezapomeňte, že instrukcí CPD se obsah HL sníží o 1. Proto je nutno od něj odečíst 7, abychom se dostali na 1. adresu dalšího záznamu, který je uložen vždy od adresy o 8 adres nižší než předchozí. Po nalezení shodnosti prvních bajtů se porovnávají další. Pokud bychom chtěli porovnávat delší řetězec, bylo by vhodnější zařadit do programu smyčku. Nejsou-li nalezeny žádné shodné bajty, program se vrátí po provedení instrukce JP PO, KONEC.

Ovšem pozor! Není zde ošetřen jeden mezní stav. Pokud je BC=0 a A = (HL), program se provedením instrukce JR Z, POR vyhne zjištění stavu P/V v instrukci JP PO, KO-NEC. Při dalším provedení instrukce CPD se obsah BC sníží o 1 na FFFFH. Tak se program neukončí a bude následovat dalších 65535 exekucí instrukce CPD! A poté znova, "navěky věkův".

Tento případ neošetření mezního stavu v programu jsem uvedl *záměrně,* abych vás upozornil na to, že může vést k velmi nepříjemným důsledkům. Mezní stavy se vyskytují prakticky v každém programu. Je na programátorovi, aby všechny mezní stavy svého programu dokázal rozpoznat, předpovědět je. Již nejednou se stalo, že opomenutý mezní stav se v profesionálním softwaru projevil třeba až po mnoha měsících provozu v nějakém podniku. A pokud jeho chybné výsledky nebyly zřejmé na první pohled, mohl nadělat v řízení nebo evidenci podniku slušnou paseku. Proto ve svých programech nikdy neopomeňte vychytat všechny potencionální zdroje mezních stavů a náležitě je ošetřete.

KAPITOLA 7

O INTERFACINGU

Věřte nevěřte, právě jste absolvovali všechny standardní instrukce Z80 kromě několika posledních, které se vztahují k příjmu dat z periferních zařízení a odesílání dat na periférie. Jde o softwarové ovládání hardwaru pomocí vstupních a výstupních portů To, jak budou tato data „vypadat", programujeme instrukcemi, které jsme si dosud probrali.

Slovo port v řadě jazyků znamená přístav. I počítačový port si můžete představit jako přístav, do něhož buď z dálky připlouvají data, kde si je vyzvedáváme, nebo jsou do něj expedována před jejich odplutím do hardwarových dálav (periférií).

Interfacing v sobě ani zdaleka nezahrnuje jen pár instrukcí pro komunikaci mezi počítačem a perifériemi. Interfacing je především vědou o hardwaru, vyžadující speciální studium. Protože tato učebnice je zaměřena na programování ve strojovém kódu, nelze se v ní hardwarem zabývat v míře vyčerpávající téma (text by byl nejméně třikrát rozsáhlejší). Povíme si jen několik základních informací, které jsou naprosto nezbytné pro pochopení funkcí instrukcí ve styku s perifériemi.

Mikropočítače komunikují s připojenými zařízeními dvěma typy stykových jednotek zvaných interface - paralelními a sériovými. Sám interface je „černá krabička", která vykonává funkce nezbytné pro úspěšný převod dat z počítače do periférie, i ve směru opačném. Vlastní vstup a výstup počítačů je konstruován jako porty, k nimž se interfacy připojují. Stejně jako interfacy, jsou i porty paralelní a sériové.

Zde je nutno poznamenat, že někdy (zvláště v četných počítačových časopisech) se nedělá rozdíl mezi slovy port a interface, což není správné. Port může být součástí interfaců, ale také nemusí. Interface obsahuje řadu aktivních i pasivních součástek a může být vybaven i svým operačním systémem. Jeho funkce spočívá především v tom, že data TRANSFORMUJE do podoby, resp. formátu, který je pro tu kterou komunikaci nezbytný. Zároveň zajišťuje synchronizaci přenosu Interfacem může být rozuměn dokonce i jen samotný software, který opět TRANSFORMUJE data na požadovaný formát.

Paralelní port může být vstupní nebo výstupní. U osmibitových počítačů má pro odesílání nebo příjem dat osmilinkovou datovou sběrnici. To znamená, že na této sběrnici můžeme v každém okamžiku pracovat současně (paralelně) až s osmi informacemi (log.1 nebo log.0 na každé lince), tedy osmi bity jednoho bajtu

Sériový port může rovněž být vstupní nebo výstupní. Datová informace jde jen po jedné lince. Přenášet jeden bajt !ze tedy jen postupně - po lince jde bit po bitu. Tyto porty vyžadují kódovací a dekódovací software, který je schopen podle daného formátu

129dat (způsobu jejich konstrukce) probíhající informaci kódovat a dekódovat. Pro-

tože se sériový přenos používá hlavně pro komunikaci na větší vzdálenosti, je nutné programově zajistit, aby v jeho průběhu nedošlo k sebemenšímu zkreslení, destrukci dat.

Pro přenos dat jsou nezbytné další doplňující linky, kterými se obě komunikující strany informují o průběhu komunikace a zároveň ji synchronizují. Např. při tisku na tiskárně jsou data (řídicí kódy pro ovládáni tisku a ASCII kódy znaků, které mají být vytisknuty) vysílána z výstupního portu počítače. Tiskárna je však přijme jen tehdy, když není zaměstnána jinými pracemi. O tom předává informace počítači právě po doplňujících linkách. Má-li tiskárna plnou hlavu vlastního tisku, sdělí, že je BUSY - zaměstnána (má napilno) a data nepřijímá. Počítač zastaví jejich odesílání na výstup. Jakmile se však objeví signál READY, který sděluje, že tiskárna už všechno „snědla", počítač okamžitě po přijetí synchronizačního pulsu znova začne chrlit data do jejího bufferu (vyrovnávací paměti). Tento rychlý přenos je synchronizován signálem STROBE, který zároveň potvrzuje platnost dat. Po naplnění bufferu daty tiskárna opět oznámí, že je BUSY a celý cyklus se opakuje, dokud má počítač co dodávat. Těmto komunikačním linkám se anglicky vtipně říká *handshake* (doslovný překlad znamená „potřásání rukou"). Volně můžeme přeložit jako „tahání za rukáv" - tiskárna a počítač se vždy zatahají za rukáv, když po tom druhém něco chtějí: Už! Dělej! nebo Chvilinku počkej!

Výhody paralelního přenosu dat jsou zřejmé - jednodušší software a minimálně osminásobná rychlost přenosu. Proč se tedy vůbec zabývat sériovým? Protože počítače vstoupily do „jednodrátové" civilizace. Jistě by bylo mnohem méně hospodárné měnit všechny existující linky (např. telefonní pro připojení modemů) na víceżilové pro každou zúčastněnou stanici. Nezbývá tedy než se spokojit s tím, co je, a počítačový vstup i výstup tomu podřídit. A jak také jinak zaznamenat data na jednu stopu magnetofonu, než sériově? Důvodů pro aplikaci sériového přenosu dat by bylo možno uvést víc.

Port je adresovatelným místem paměti, které po přijetí bitů na svém vstupu dává potřebnou logickou úroveň na výstupu. Ta je předána sběrnici, na niž se připojuje patřičný interface. Nesmíte se však domnívat, že komunikace počítače s externími zařízeními probíhá jen z něj vyvedenou sběrnicí. Např. už sám výstup pro připojení obrazovky obsahuje přetransformovaná, obrazovým interfacem zpracovaná obrazová data. Tento interface je přímo v počítači (nebo monitoru). Podobně je tomu s výstupem a vstupem pro záznam a čtení dat z pásku nebo klávesnice. Interface není nutně jen samotný hardware; jeho součástí může být i software (operační systém) s implementovanými obslužnými funkcemi.

Instrukční soubor Z80 umožňuje adresovat maximálně 256 vstupních i 256 výstupních portů. Z tohoto počtu se obvykle využívá jen část. Pro ostatní prostě není žádné využití. Umíte si snad představit svůj mikropočítač napojený na řekněme 230 periferních zařízení? Porty jsou osmibitové. Každý z těchto bitů každého portu může ovládat nějakou systémovou funkci nebo o nich podávat informace. To záleží jen na konstrukci počítače a interfaců. Abyste je mohli plně ovládat, potřebujete znát i tzv. firmware, v němž výrobce *uvádí* nezbytné informace o ovládání hardwaru svého počítače, tedy i o tom, jak jeho jednotlivé porty pracují.

Z boje o standardizaci spojení mikropočítačů s periférierni nakonec vyšly vítězně dva interfacy - paralelní Centronics a sériový RS232 (a jejich varianty). Proto pro připojeni těchto interfaců jakékoli hardwarové úpravy nejsou nutné, vše by mělo fungovat na-

poprvé. Samozřejmě, že periférie musí být vybavena pro komunikaci s daným typem interfaců. Při programovém ovládání hardwaru jde zase jen o známé posílání binárních jedniček a nul na pravé místo v pravý čas a ve správné posloupnosti.

INSTRUKCE VSTUPNÍ IN A VÝSTUPNÍ OUT

Vstupními instrukcemi odebíráme log. stav osmi bitů zvoleného vstupního portu. Instrukcí IN A, (port) převedeme po datové sběrnici obsah portu (jeho číslo je v intervalu 0..255) do bitů reg. A. Instrukcí IN r, (C) (r je jeden z registrů A, B, C, D, E, H nebo L) bude převeden obsah portu adresovaného obsahem registru C do reg. r. Chceme-li např. zjistit stav nějaké periférie, která o sobě posílá informace do počítače přes vstupní port 255, pak tyto informace můžeme dostat do reg. A nebo r těmito způsoby:

```
IN A, (FFH)        nebo        LD C, FFH
                               IN H, (C)
```

Jednoduché, že? Nebýt ovšem toho, že v průběhu čtení stavů portů někdy musí být blokováno přerušení, nebo se čeká na puls přerušení, kterých může být řada a jimiž se spouštějí obslužné rutiny pro zpracování dat, atd ... O něčem z toho až dále.

Vstupní **instrukce IN** pracuje v tomto sledu:

1. Na nižších 8 bitů adresové sběrnice uloží číslo portu a na vyšších 8 bitů adresové sběrnice uloží v případě instrukce IN A, (port) předchozí obsah reg. A; při IN r, (C) předchozí obsah reg. B.
2. Aktivují se signály IORQ a RD (linky Z80).
3. Po datové sběrnici převede do reg. A, ev. jiného, obsah adresovaného portu.

Zvláštnosti popsané v bodu 1 (reg. B jako vyšší bajt adr. sběrnice) se při komunikaci využívá různým způsobem. Záleží na zapojení hardwaru. Vyšším bajtem adresové sběrnice můžeme třeba zjišťovat stav nějakého jiného portu současně s portem adresovaným nižším bajtem této sběrnice. Nebo těmito vyššími linkami poslat nějakou informaci na periférii apod.

Blokové vstupní instrukce mají svou analogii v instrukcích LDI, CPI, LDIR, CPIR, LDD, CPD, LDDR a CPDR. Samozřejmě, že s některými odlišnostmi. Všechny se inicializují takto:

reg. B - počet bajtů odebíraných z portu
reg. C - číslo portu (neboli jeho adresa)
reg. HL - 1. adresa uložení odebraného bajtu

Průběh instrukce INI *(Input-Increment):*

1. Obsah reg. B se sníží o 1 (ve smyslu DEC B).
2. Z portu adresovaného obsahem reg. C se odebere bajt a uloží se na adresu (HL).
3. Obsah reg. HL se zvýší o 1 (ve smyslu INC HL).
4. Když je obsah reg. B nulový po kroku 2, pak Z=1. N = 1 vždy.

Průběh **instrukce INIR** *(Input-Increment-Repeat):*

1. až 3. jako INI
4. Není-li obsah reg. B nulový, vše se opakuje od bodu 1. Průběh končí, když B = 0. Pak program pokračuje další instrukcí. Stav SI tentýž jako u INI.

Analogicky probíhají **instrukce IND a INDR** (D znamená *Decrement).* Rozdíl je jen v tom, že obsah reg. BL se snižuje o 1 (ve smyslu DEC HL).

Pro instrukce výstupní platí analogie s instrukcemi vstupními. Rozdíl je v tom, že data z portu neodebíráme, ale na port je posíláme. Přitom se neaktivuje signál RD (READ - čti), ale WR (WRITE - zapiš).

Výstupní instrukce	analogická ke	vstupní instrukci
OUT (port), A		IN A, (port)
OUT (C), r		IN r, (C)
OUTI		INI
OTIR		INIR
OUTD		IND
OTDR		INDR

Oproti vstupním je u výstupních blokových instrukcí jedna změna v pořadí sledu probíhajících operací — nejdříve se provádí bod 2, pak analogicky bod 1 (viz výše).

Tak např. instrukce OUT (port), A odešle po datové sběrnici obsah reg. A na adresovaný port — v případě OUT (FDH), A na port 253 apod. Ostatní rozdíly mezi instrukcemi IN a OUT jsou čistě hardwarové

INSTRUKCE PŘERUŠENÍ DI a EI

V každém počítači probíhá určitý počet synchronizovaných, většinou cyklických dějů, jimiž je řízen tok informací. Probíhají skokově, přerušovaně, prolínají se dle stanovených algoritmů svých průběhů. Tak třeba se stalo dobrým zvykem, že každý počítač po připojení ke zdroji napřed provede kontrolu svého hardwaru. Když tento test najde někde chybu, počítač se „zakousne" a nedá se s ním dělat už nic jiného, než chybu najít a opravit. Pokud test proběhne v pořádku, celý hardware počítače se začne „točit" podle povelů obsažených v jeho operačním systému. I vstupní testy jsou součástí operačních systémů, které se u různých typů počítačů velmi liší. Snaha o standardizaci systémem MSX zkrachovala. Přišli s ním Japonci a mysleli to dobře. Jenže MSX se stal brzdou vývoje, závažím, které měsíc po měsíci těžklo. A tak máme v operačních systémech slušný zmatek. I když každý na to jde trochu jinak, mají všechny něco společného.

Každý systém musí stále testovat klávesnici — zda je některé z tlačítek stisknuto, a když ano, tak které. Stále je nutno tvořit televizní obraz zobrazením obsahu obrazové paměti počítače, nepřetržitě probíhá řada testů a změn.

Např. u mikropočítače ZX Spectrum je klávesnice testována každou padesátinu vteřiny. To je zároveň čas potřebný pro tvorbu jednoho TV půlsnímku. Všechny průběžné

procesy ovládané jedním mikroprocesorem nemohou probíhat současně. Jejich cyklický sled je umožněn opakováním interního signálu přerušení. Jeho zablokováním instrukcí DI *(Disable Interrupt)* vyřadíte z činnosti tu část operačního systému, která je řízena přerušením. Většinou tím však nemusíte vyřadit např. tvorbu obrazu. To záleží na konstrukci počítače i na jeho programovém vybavení. Zablokované přerušení uvolňuje instrukce EI *(Enable Interrupt).* To se však nevztahuje k vnějšímu, nemaskovatelnému přerušení (viz dále). Funkci DI zařazujeme do programu zásadně vždy, kdykoli by maskovatelné přerušení přineslo destrukci výsledku prováděných operací — především v případě časově kritických operací, jako je záznam a čtení dat při komunikaci se záznamovým zařízením, přenos dat po modemové lince apod. Vstup nevítaného přerušeni do takové komunikace by způsobil nenapravitelnou deformaci přenášených dat.

Kdykoli potřebujeme znovu uvolnit přerušeni, zařadíme do programu instrukci EI. K ní se váže speciální instrukce návratu z obslužné rutiny maskovatelného přerušení RETI. Její použití je nutné jen v případech komunikace se speciálními periériemi (řazené SIO, PIO obvody apod.). Jinak lze místo ní zařadit nám už dobře známou RET. K návratu z nemaskovatelného přerušení slouží instrukce RETN.

Módy přerušení a jejich instrukce

Napřed si probereme dva základní typy přerušeni (nikoli módy!) - maskovatelné a nemaskovatelné. Představte si sled těchto dějů:

1. Čtete si knížku.
2. Začne zvonit telefon.
3. Odložíte knížku, zvednete telefon a začnete s někým mluvit.
4. Vtom zazvoní zvonek u dveří.
5. Odložíte sluchátko a jdete otevřít.
6. Vyřídíte záležitost u dveří a vrátíte se.
7. Zvednete sluchátko a dokončíte hovor.
8. Vezmete si knížku a budete pokračovat ve čtení.

Vaše čtení bylo přerušeno telefonem. Telefonní hovor byl přerušen zvonkem u dveří. Vždy jste reagovali tak, že jste předchozí činnost přerušili a šli vstříc novému podnětu. Po posledním přerušení jste pak ve zpětném sledu vyřizovali, co s sebou jednotlivá přerušení přinesla. Znázorníme-li si uvedené hlavní činnosti písmeny A, B, C, pak (už bez dalších přerušení) jste akce dokončili v pořadí C, B, A, přičemž v A (čtení) setrváváte s nadějí, že už nebudete rušeni. Ale když se nějaké přerušení opět objeví, budete je akceptovat.

Je tu však ještě jedna možnost - když nechcete zvednout telefon ani jít otevřít dveře, prostě nebudete na podněty reagovat. Budete se „maskovat". Nepřipustíte přerušení své momentální činnosti. Uvedená možnost vymezila pojmenováni tohoto druhu přerušení adjektivem maskovatelné. Po provedení instrukce DI mikroprocesor ignoruje přicházející přerušení a nereaguje na ně. Dělá, jako by „nebyl doma". K tomu, aby přerušení přijal a reagoval na ně, ho zase přimějete instrukcí EI.

Co však, když si čtete a začne hořet dům? Zřejmě budete reagovat okamžitě. Tomuto druhu přerušení se říká nemaskovatelné Mikroprocesor je nemůže ignorovat,

a kdykoli se objeví, okamžitě na ně reaguje. Toto vnější přerušení se užívá především při vzniku havarijních situací (výpadek proudu, porucha nějakého zařízení apod.), resp. kdykoli potřebujeme dát přednost provedeni funkce s nejvyšší prioritou.

Nyní si shrneme obecné způsoby řízení mikroprocesorů v interfacingu. Jsou dva:

1. **Softwarové řízení** prostřednictvím programu, který nepravidelně nebo cyklicky zjišťuje, zda periférie vyžaduje nebo povoluje styk.

2. Soft-hardwarové řízení programem, který:
   a) obsahuje módy přerušení:
      V průběhu provádění instrukce mikroprocesor zároveň zjišťuje, zda se neaktivuje vstup přerušení. V případě že ano, dokončí právě prováděnou instrukci a poté vstoupí do daného módu přerušení.
   b) umožňuje provedení DMA procesu:
      DMA *(Direct Memory Access* - přímý přístup do paměti). Jde o proces velmi rychlého přenosu bloků dat z periférie (i vedlejší paměti) do počítače a naopak. Rychlost přenosu realizovaného speciálními obvody je cca 1 MB za vteřinu. Během tohoto přenosu je mikroprocesor vyřazen z činnosti; po jeho skončení v ní pokračuje.

Uvedené způsoby řízení si převeďte na analogii s obsluhou telefonu. Pod bodem 1. budete stále zvedat a pokládat sluchátko, abyste zjistili, zda někdo nevolá (zvonek je vypnut). V bodu 2.a) reagujete na vyrušení zvonkem telefonu. U 2.b) máte k telefonu připojenou tzv. automatickou sekretářku. Přenos probíhá mimo vaše vědomí. Obsah vzkazu si přehrajete, až budete chtít nebo potřebovat.

Z hlediska hardwarového se externí zařízení dožadují komunikace přerušením ve třech provedeních:

- **Přerušení po jedné lince:**
  Jakmile CPU zjistí na svém vstupu signál přerušení, ihned přeruší momentální činnost a věnuje se periférii. Tato metoda je nejrychlejší, když je k počítači připojeno jen jedno externí zařízení. Při větším počtu periférií se většinou musí zjišťovat, která z nich si přerušení vyžádala. K jedné lince může být připojeno libovolné množství externích zařízení.

- **Přerušení po více linkách**
  Každá periférie má svou linku přerušení, takže mikroprocesor nemusí zjišťovat, která z nich se dožaduje komunikace. Počet připojitelných periférií závisí na počtu linek přerušení mikroprocesoru (mívají 4 takové linky) - netýká se Z80.

- **Vektorovaná přerušení**
  Spolu se signálem přerušení vyšle externí zařízení na sběrnici buď jednobajtový operační kód instrukce k jejímu okamžitému provedení (IM 0), nebo část vektorové adresy, na niž je ihned převedeno programové řízení (IM 2). Rutině, která pak bude provedena, se říká obslužná rutina přerušení.

Nemaskovatelné přerušení je identifikováno na vstupní lince NMI Z80. Aktivuje se úrovní log. 0. Tehdy Z80 přeruší svou dosavadní činnost a okamžitě převede programové řízení na adresu 0066H, kde začíná obslužná rutina tohoto přerušení. De facto je signál NMI analogií instrukce CALL 0066H. De iure je však „pachatelem" periferní zařízení. Návratu z rutiny slouží instrukce RETN.

Mikroprocesor je vybaven dvěma klopnými obvody (flip-flopy) pro blokování či otevření přijetí signálu maskovatelného přerušení IFF1 a IFF2 *[Interrupt Flip — Flop).* Jejich log. stav je přímo nastavitelný instrukcemi Dl a El. Dl je nuluje (blokuje), El nastavuje na log. 1 (uvolňuje přerušení). Okamžikem přijetí maskovatelného přerušení se stav obou IFF automaticky překlopí do log. 0. IFF2 slouží k uchování log. stavu IFF1 v případě, kdy se aktivuje signál NMI. Instrukce RETN pak převede stav IFF2 do IFF1. Log. stav IFF2 lze převést do indikátoru P/V provedením instrukce LD A,I nebo LD A,R a testovat podmínkami PE (0) a PO (1). Stav IFF1 testovat nelze.

Módy maskovatelného přerušení jsou tři. Každý z nich má svou instrukci: IM 0, IM 1, IM 2 *(Interrupt Mode).* Jejím provedením se aktivuje jí určený mód přerušení (nastavením kombinace na dalších **flip—flopech** mikroprocesoru Z80- IMFa a IMFb).

Mód 0 — periférie pošle na datovou sběrnici jeden bajt, který Z80 čte jako instrukci. To však neznamená, že instrukce předaná počítači může být jen jednobajtové Kombinací bajtu s jinými lze vytvořit instrukci vícebajtovou.

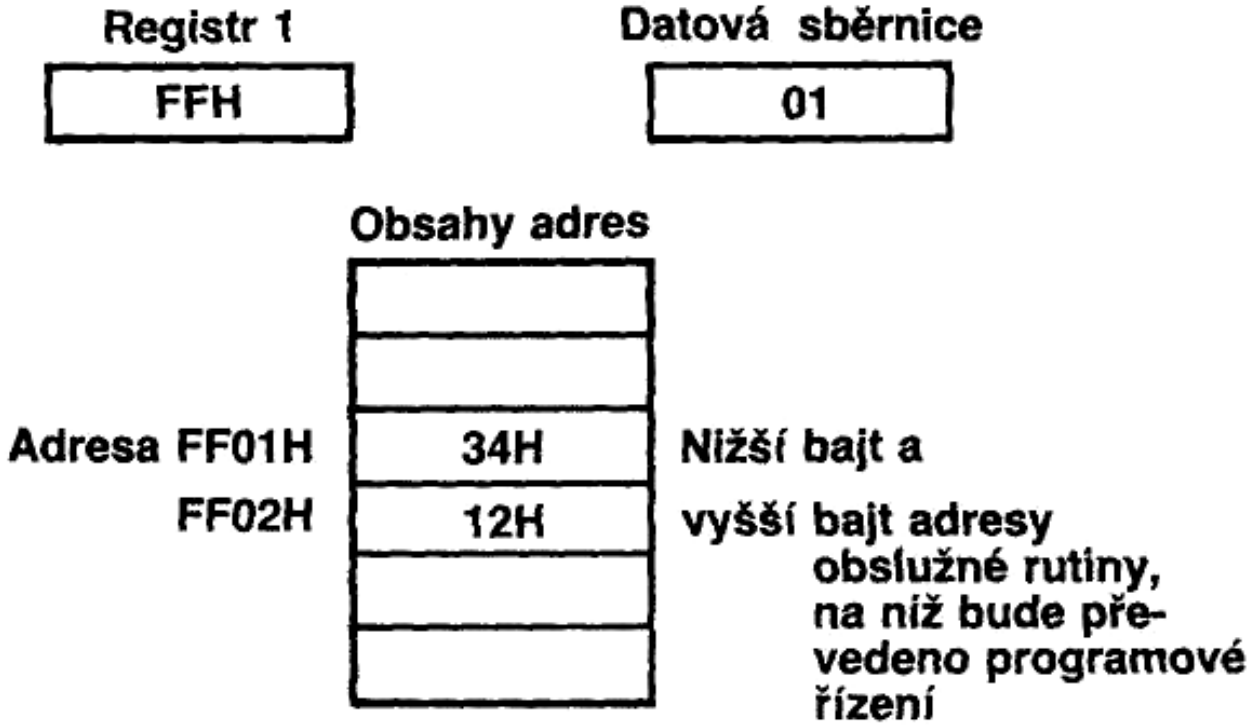
Mód 1 je operačním ekvivalentem instrukce RST 38H.

Mód 2 je o něco málo komplikovanější než ostatní. Periférie vyšle na datovou sběrnici jeden bajt, který tvoří nižší část adresy vektorové tabulky adres obslužných rutin. Vyšší část adresy je v reg. I (zvaném *Interrupt Vector-* vektor přerušení). Na takto formované adrese leží nižší bajt a na adrese o 1 vyšší pak vyšší bajt adresy obslužné rutiny tohoto přerušení. Jinými slovy - na vektorové adrese a adrese o 1 vyšší leží dva bajty pro doplnění adresy instrukce CALL XXXX, která je ihned poté (automaticky) provedena. Pokud neměníme inicializovaný obsah reg. I (jeho obsah budiž třeba AAH), pak při různých obsazích bajtů posílaných z periférií na datovou sběrnici můžeme mít v počítači až 256 obslužných rutin. Bajty jejich 1. adresy budou na adresách AA00H a AA01H, poslední na adresách AAFFH a AB00H. Samozřejmě, že i reg. I můžeme v průběhu programu měnit. Ale takové množství (65536) obslužných rutin by se nám do počítače ani nevešlo.

Pro lepší pochopení funkce IM 2 uvedu příklad. Dejme tomu, že potřebujeme, aby poté, co se po přerušení na datové sběrnici objeví bajt 01 (odeslaný z periférie do počítače), byla spuštěna rutina od adresy 1234H. Napřed musíme mít v reg. I vyšší bajt adresy vektorové tabulky, na níž leží nižší bajt adresy 1234H, tedy 34H. Zvolme třeba FFH. Tak na adrese FF01H bude bajt 34H, na adrese FF02H bajt 12H. Přenos FFH do reg. I provedeme takto:

```
LD A, FFH
LD I, A
```

Do místa programu, od něhož povolíme přerušení v módu 2, zařadíme instrukci IM. 2. Nyní, kdykoli si bude chtít (nebo potřebovat) periférie s námi popovídat, vyšle na flip-flop IFF1 signál přerušení. Protože jde o přerušení maskovatelné, musí být uvolněno; nesmí tedy být blokováno působením instrukce DI. Pokud je tomu tak, musíme přerušení uvolnit instrukcí EI. Je-li vše v pořádku, na datové sběrnici se objeví bajt s obsahem 01. K němu se připojí obsah reg. I a spolu vytvoří adresu FF01H. Z této vektorové adresy mikroprocesor přenese do nižší poloviny reg. PC bajt 34H a z adresy o 1 vyšší (FF02H) bajt 12H, který uloží do vyšší části reg. PC. Na takto vytvořenou adresu 1234H bude okamžitě převedeno programové řízení. Adresa návratu byla uložena do zásobníku již před realizací přerušení. Po exekuci obslužné rutiny přerušení od adresy 1234H bude (instrukcí RETI) proveden návrat tam, kde byl program přerušen (odtud pak bude pokračovat dál). Schematicky:



Je-li uvolněno přerušení a počne se provádět jeho obslužná rutina, není nutné ji chránit před vpádem dalšího maskovatelného přerušení instrukcí DI. Přijetím signálu přerušení před vstupem do obslužné rutiny se přerušení automaticky zablokuje. Chcete-li je po odbavení obslužné rutiny opět uvolnit, rutinu zakončete instrukcemi EI a RET, resp. EI a RETI. Přitom je dobré vědět, že přerušení bude uvolněno až po vykonání instrukce následující za EI (zde po RET, resp. po RETI), tedy nikoli hned po EI. Oproti tomu instrukce DI blokuje přerušení ihned po svém provedení. Rovněž nezapomeňte uschovat obsahy všech registrů instrukcí PUSH na začátku každé obslužné rutiny! Na jejím konci je zase přeneste zpět do registrů instrukcí POP. Nikdy totiž nemůžete vědět, v jakém místě programu k přerušení dojde. V případě, kdy je přerušen cyklický běh instrukce blokového přenosu nebo prohledávání dat s opakováním, aniž byl dokončen, po provedeni obslužné rutiny přerušení pokračuje běh instrukce tam, kde byl přerušen.

INSTRUKCE ZASTAVENI HALT

Tato instrukce zastaví běh programu až do momentu, kdy se objeví jakékoli přerušení - v případě maskovatelného ale jen tehdy, je-li přerušení uvolněno (EI). Po provedené operaci s přerušením související pak program pokračuje (není-li určeno jinak) instrukcí umístěnou hned za HALT. Při zablokovaném přerušení (DI) se běh programu obnoví jen při aktivaci linky NMI (nemaskovatelném přerušení). Pokud tedy použijete tuto zvláštní instrukci po DI a nemáte možnost aktivovat NMI, program se vám zastaví navždy. Proto pozor při jejím užití. Instrukce HALT během svého působení provádí stále dokola „nopy" (viz dále), takže navenek se nic neděje.

EXPERIMENT poslední

Ukázka užití IM 0 a tří obslužných rutin v IM 2. Bohužel si ji na počítači nemůžete demonstrovat, pokud k němu nemáte připojena tři experimentální periferní zařízení

```
; Uloženi adres obslužných rutin přerušení do vektorové tabulky
; těchto adres (tabulka je na adresách FF00H-FF05H)
;
                DEFW:   1234H           ;2 bajty na adresách FF00H a FF01H
                        2345H           ;2 bajty na adresách FF02H a FF03H
                        3456H           ;2 bajty na adresách FF04H a FF05H
;
; ———————————————————————————————————————————————————————————————————
;
                START:  DI              ;Blokování maskovatelného přerušení
                        LD A, FFH       ;Vyšší bajt vektorové adresy do reg. A
                        LD I, A         ;Jeho přenos do reg. 1
                        EI              ;Uvolnění přerušení
                        IM 0            ;Nastavení na mód přerušení 0
                        HALT            ;Čekání na signál přerušení
                        IM 2            ;Nastavení na mód přerušení 2
                        HALT            ;Čekání na signál přerušení
                        JP PROGRAM      ;Skok na adresu PROGRAM
;
; ———————————————————————————————————————————————————————————————————
;
;OR1 začíná na adr. 1234H
;
                OR1:    PUSH AF         ;Uložení obsahů všech reg do zásobníku
                        PUSH BC         ;Mask. přerušení je automaticky bloko-
                        PUSH DE         ;váno předchozím signálem přerušení
                        PUSH HL
                        PUSH IX
                        PUSH IY
```

```
;
; Zařazení jakékoli subrutiny,                    která bude provádět požadovanou
; funkci při aktivaci přerušeníz periférie 1
;
                        POP IY          ;Reinicializace obsahů registrů
                        POP IX
                        POP HL
                        POP DE
                        POP BC
                        POP AF
                        EI              ;Uvolnění přerušení
                        RETI            ;Návrat z obslužné rutiny přerušení
;
; ---------------------------------------------------------------------
;
; OR2 začíná na adr. 2345H
;
                OR2:...                 ;Napřed uložit reg. do zás. jako u OR1
                        EI

; Zařazení obslužné subrutiny pro přerušení z periférie 2
; Přenos uložených bajtů ze zásobníku zpět do registrů (POP)
;
                        RETI            ;Návrat
;
; ---------------------------------------------------------------------
;
; OR3 začíná na adr. 3456H
;
                0R3:...                 ;Opět uložit reg. do zás. jako u OR1
                        EI
;
; Zařazení obslužné subrutiny                     pro přerušení z periférie 3
; Přenos uložených bajtů ze zásobníku zpět do registrů (POP)
;
                        RETI            ;Návrat
```

KROK poslední: K počítači jsou připojena tři periferní zařízení, číslovaná 1, 2 a 3. Na začátku rutiny START je blokováno přerušení (DI), aby případně nevstoupilo do děje před potřebnou inicializací registru I. Dále je přerušení uvolněno (EI) a nastaven IM 0. Jakmile se na IFF1 objeví signál přerušení z některé periférie, provede se instrukce, jejímž operačním kódem bude bajt periférií odeslaný na datovou sběrnici - mohou to být např. jednobajtové instrukce RST, IM, LD, logické, aritmetické atd. Po vykonání instrukce (pokud nás nezavede jinam, bez návratu na další instrukcí rutiny START) je mód přerušení změněn na IM 2. Když se jednotlivá periferní zařízení (PZ) dožadují spojení v tomto módu přerušení, posílají na datovou sběrnici tyto bajty:

PZ1 00
PZ2 02
PZ3 04

Tak třeba když si komunikaci vyžádá PZ2, na datové sběrnici bude bajt 02. S obsahem reg. I bude sestavena vektorová adresa FF02H. Z této adresy a adresy o 1 vyšší budou přeneseny 2 bajty do reg. PC. Tak přejde programové řízení na adresu 2345H. Na ni je obslužná rutina OR2, která po vykonání svých operací provede instrukci návratu RETI a vrátí se tam, kde bylo přerušení aktivováno.

Zde si povšimněte toho, že zatímco v rutině OR1 je přerušení zpočátku zablokováno (rutina se tedy provede celá, pokud se neobjeví nemaskovatelné přerušení), OR2 a OR3 mohou být během svého provádění kdykoli přerušeny maskovatelným (a samozřejmě i nemaskovatelným) přerušením. Jednotlivé rutiny (v jejich prolnutí) budou pak provedeny ve sledu analogickém s výše uvedeným příkladem čtení, telefonu a zvonku u dveří. Tomuto řazení exekucí obslužných rutin se anglicky říká *nested* (vysl. nestyd). Svou vzdálenou analogii má v ukládání a odebírání dat ze zásobníku - první dovnitř, poslední ven. Protože před vstupem do obslužné rutiny je z reg. PC do zásobníku uložena adresa návratu k instrukci, po jejímž provedení bylo přerušení přijato, je nutno věnovat patřičnou pozornost tomu, aby nám při užití vnořeného prolínání rutin zásobník nakonec neprolnul do programu.

V uvedeném experimentu je nezbytné do detailu promyslet práci celé rutiny, hlavně přenos dat mezi zásobníkem a registry. Někdy je nezbytné takový „agregát rutin" podřídit systému priorit přerušení.

Prolínání obslužných operací umožňuje např. zapojení typu *daisy chain* (čte se dejzy čejn). Poetický název „věneček ze sedmikrásek", v tomto případě spíše řetězení exekucí obslužných rutin, označuje řazení periférií do řetězce podle stupně jejich priority. Periférie s vyšší prioritou může na čas potřebný pro svou obsluhu přerušit obsluhu periférie s prioritou nižší. Pro návrat z rutin pak musíme používat instrukci RETI, která na datovou sběrnici umístí kód priority obslouženého zařízení. Čekají-li ve frontě zařízení s nižší prioritou, jsou postupně obsloužena po skončení obsluhy zařízení s vyšší prioritou. Jakmile však během této doby přijde žádost o přerušení od zařízení s vyšší prioritou, dostane okamžitou přednost. Atd ...

V obou předchozích případech jde o specifické soft-hardwarové řízení komunikace, které je nutno řešit případ od případu. Zde nezbývá než odkázat na odbornou literaturu.

Vraťme se k našemu programu. Po prvním maskovatelném přerušení v IM 2 (čeká se na něj opět po instrukci HALT), následném provedení patřičné obslužné rutiny a návratu z ní zpět do rutiny START, je provedena instrukce skoku do libovolného programu (JP PROGRAM), který někde v počítači máme. Jeho průběh bude přerušen vždy, kdykoli si to některé z PZ bude přát (samozřejmě jen tehdy, bude-li přerušení uvolněno).

Připojíme-li nyní na línku NMI další periférii, bude aktivací této linky přerušen chod programu vždy a programové řízení bude převedeno na adresu 0066H. Pro doplnění - nejvyšší prioritu ve struktuře přerušení má aktivace linky BUSREQ, kterou se zajišťuje spolupráce Z80 např. s koprocesory. Zde však nejde o přerušení ve smyslu volaní obslužné rutiny, ale spíše o „zmrazení" činnosti mikroprocesoru.

Může se vám zdát, že důsledky aktivace NMI a IM 1 jsou značně chudé - jen skok na adresy 0066H nebo 0038H. Nezapomeňte však, že rutiny umístěné na těchto adresách můžeme podmíněně větvit v závislostech, které si sami stanovíme. Proto se i výsledky, které provedení rutin přinese, mohou vždy něčím lišit. Pokud máte počítač, který má na spodních adresách pevně zabudovanou romku, budete v jejich využití omezeni - ale jen do té doby, než využijete možnosti stránkování paměti pro připojení vedlejší paměti s vaším vlastním operačním systémem.

V programu uvedená komunikace je jednosměrná, navíc velmi chudá. Je tu jen jedno prosté „zatahání za rukáv", dokonce bez přenosu dat mezi počítačem a PZ. V praxi vše probíhá členitěji. Především sama komunikace bývá obousměrná. Firma Zilog vyrábí pro interfacing se Z80 speciální integrované obvody PIO a SIO (programovatelné porty pro paralelní a sériový přenos dat).

Prakticky jakýkoli integrovaný obvod se může stát součástí interfaců nebo periférie. Podobně jako programování je stavba hardwaru činností tvůrčí. Vše jde vždy řešit různě - standardně, složitě, jednoduše, ale i chytře, neotřele - a o to by mělo jít především. Právě v tom tkví výjimečný úspěch některých počítačů. Jsou totiž po stránce hardwarové (spolu s jejich nápaditě komponovaným operačním systémem) řešeny lépe, zajímavěji než jejich předchůdci, rozšiřují možnosti využití techniky, kterou reprezentují. Jsou-li dobře zabezpečeny i po stránce marketingu, začneme se brzy dozvídat, že obsazují první příčky počítačových žebříčků obliby i prodejnosti.

Pokud někdy budete chtít vymýšlet a konstruovat přenosová zapojení, začněte přenosem paralelním. Sériový je neporovnatelně komplikovanější. V každém případě se do ničeho nepouštějte, dokud nezvládnete teorii hardwaru. Vyhnete se tak pohledu na dýmající počítač.

Na konec jsme si nechali instrukci nejjednodušší:

NOP

Znamená *No Operation,* tedy žádná operace. Při inicializaci mikropočítače jeho operační systém těmito „nopy" obvykle vyplní celou volnou paměť RAM. HD kód instrukce NOP je 00. I když na adrese obsahující samé nuly vlastně nic není, nelze to brát tak doslova. Provedení této instrukce, spočívající v jejím dekódování, zabere 4 takty. Z toho vyplývá i její užití v časovacích rutinách, kde pomocí ní ladíme časově kritické operace. Její provedení nemá žádný vedlejší účinek.

POZNÁMKA K NESTANDARDNÍM INSTRUKCÍM Z80

Jedním z cílů konstruktérů mikroprocesoru Z80 bylo, aby „uměl" i jiné instrukce než ty, které jsou uvedeny v instrukčním souboru Z80. Bohužel však nedokázali opravit chyby, pro které jim další instrukce nechtěly spolehlivě pracovat. Firmy Zilog a Mostek nakonec chyby opravily, ale ostatní producenti Z80 už nikoli. U Z80 obou uvedených výrobců je možné operovat i s polovinami indexových registrů, jako by šlo o registry párové. Ostatní nestandardní instrukce jsou vedlejším efektem po minimalizaci logického

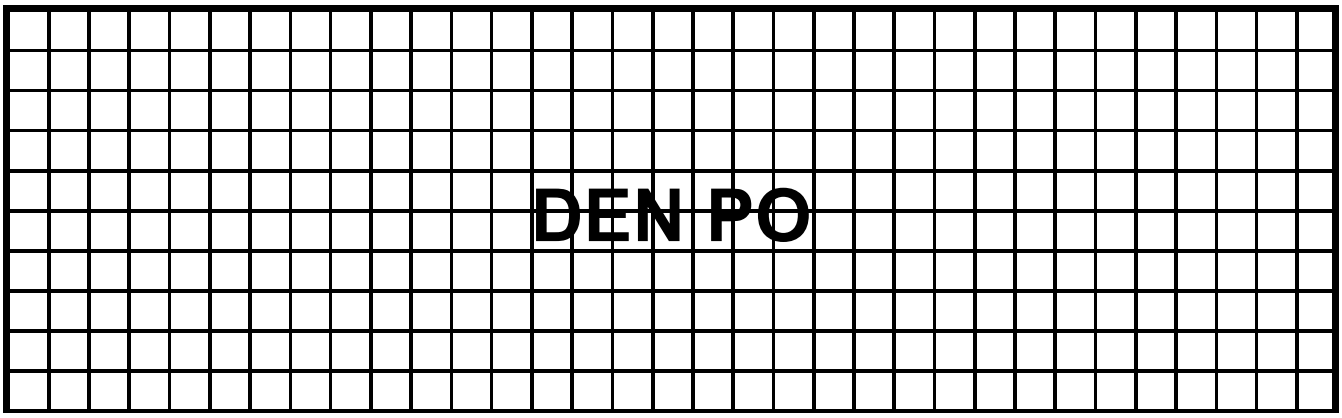
obvodu Z80

140Protože tyto instrukce nejsou zahrnuty v původním instrukčním souboru, označuj se jako nestandardní, ale i „tajné" a dokonce „ilegální". Pochopitelně nebývají implementovány v generátorech a monitorech strojového kódu. Pokud byste je chtěli ve svých programech použít, museli byste uložit do paměti jejich strojový kód (např. jako DEFB). Ale i kdyby fungovaly ve vašem počítači, vůbec to neznamená, že budou fungovat ve všech. Mívají i zvláštní vedlejší efekty (týká se SI), které mohou do programu vnést víc zmatku než užitku. Rozhodně neuděláte žádnou chybu, když s těmito exoty nebudete ztrácet čas. Z uvedených důvodů se jim tato publikace nevěnuje

3. ČÁST

DEN PO

PŘÍLOHA



„Včera jsem to všechno dočetl a doexperimentoval, je to fajn, ale ještě něco bych..."

„ ...víc už neříkej. Čekal jsem, že se objevíš přesně den poté. Ještě mám v dobré paměti svůj „den po". Jenže to už je úplně jiná kniha — snad někdy někde vyjde, ale její stránky budeš muset z větší části psát sám svou vlastní programátorskou praxí. Abys však ode mne nešel s prázdnou, připravil jsem pro tebe několik praktických rad, které ti mohou být v další práci prospěšné."

*** Informuj se o tom, který překladač assembleru a monitor jsou pro tvůj typ počítače nejlepší. Udělej vše pro to, abys je získal (i s manuálem!). Nauč se s nimi pracovat tak, abys manuál už vůbec nepotřeboval - obvykle to dá trošku zabrat, ale budeš-li pilný, nemělo by ti to trvat déle než 2-3 týdny. Začínající programátoři se často naučí zacházet jen s monitorem, který je „čtivější". Tím si však stelou Záhořovo lože. Programování samotným monitorem je velmi zdlouhavé a zvyšuje možnost „tvorby" chyb. Assemblerové návěští jsou prostě k nezaplacení. Jakmile si na svůj překladač a monitor zvykneš, pracuj jen s nimi. Všechny umějí přibližně totéž, ale mívají odlišné ovládání. Proto je zbytečné po nich pokukovat. Jen v případě, že by se pro tvůj počítač objevila nějaká softwarová hvězda, přeorientuj se (ale radši dvakrát měř...).

*** Každý kousek vytvořeného programu si nahraj před tím, než vyzkoušíš jeho funkci. Počítej s tím, že většinou ti nic nebude fungovat napoprvé. Tady se riskovat nevyplatí. Chyba v programu z 99 procent znamená jeho odchod do věčných lovišť!

*** Stále si veď dokumentaci vytvářených programů. Jednak ti usnadní orientaci při práci, a pak, když se budeš chtít k některému z nich po čase vrátit, bez dokumentace ho budeš celý zdlouhavě „luštit", protože se v něm už nevyznáš. Nejlepší dokumentací je komentovaný výpis a vývojový diagram programu. Být pečlivý v archivaci znamená šetřit svůj čas i zdraví. Ztracený „papírek s důležitými poznámkami" tě snadno přivede k zuřivosti.

*** Dávej pozor na záludnosti, které mohou vyplynout z assemblerových dvojsmyslů. Píšeš-li si nějaký kousek programu tužkou na papír, buď pečlivý. Třeba LD B,0A (i když by správněji mělo být LD B, 0AH; používá se i forma LD B, #0A) budeš vždy chápat jako přenos čísla 0AH do reg. B. Napíšeš-li v roztržitosti jen LD B, A, může se stát, že později budeš zápis chápat jako přenos obsahu reg. A do reg. B. Když takovou chybu přeneseš do programu, těžko ji pak budeš hledat. Proto HD obsah jednoho bajtu zapisuj vždy jako číslo dvoumístné.

*** Pokud ještě nemáš tiskárnu, pořiď si ji co nejdříve. Programování bez ní je o poznání obtížnější a zdlouhavější. Krátké výseky výpisu programu na blikající obrazovce ti nepoví tolik, co výpis z tiskárny. O jeho přehlednosti ve srovnání s dokumenty popsanými a proškrtanými propisovačkou ani nemluvě.

*** Jedním z největších kamenů úrazu při vlastním programování je zásobník, do něhož se ukládají a z něj odebírají adresy návratů. Do něj samozřejmě můžeme ukládat i leccos jiného a vůbec s ním všelijak operovat. Má-li tvůj program už slušnou délku (pár set bajtů) a vzdorovitě krachuje, i když se zdá být v pořádku, prober si pečlivě (ale opravdu pečlivě!), co se odehrává v zásobníku. Velmi často najdeš chybu právě tam.

*** S tím souvisí i ztráta orientace v přenosech parametrů mezi rutinami, subrutinami i jinde. U programu ve strojovém kódu stačí jeden chybný přenos a dílo zkázy je hotovo.

*** Vysoký stupeň obezřetnosti vyžaduje ošetření mezních stavů, které obsahuje prakticky každý program. Jde o programové situace s nízkým až velmi nízkým stupněm pravděpodobnosti jejich výskytu. Chyby, které přinášejí, se nemusejí nutně projevit hned, ani nemusejí být příliš patrné. Mohou, ale nemusejí se řetězit. Neošetřená místa programu, která jsou potencionálními zdroji takovýchto chyb, můžeme nazvat „studenými spoji" softwaru.

***Další časté chyby bývají zaklety v ležérním přístupu ke stavovým indikátorům. Až příliš často se testuje netestovatelné - např. indikátor Z nebo CY při DEC a INC párových registrů nebo CY při DEC a INC jednobajtových registrů atd. Na druhou stranu právě toho můžeš využít v momentech, kdy po operaci, která ovlivňuje CY, zařadíš potřebný sled instrukcí, které CY neovlivňují a jeho test provedeš až za nimi. Pokud se nenaučíš vliv instrukcí na indikátory nazpaměť (což by bylo nejlepší), měj při programování po ruce tabulku, která tyto vztahy přehledně uvádí. Druhou, i když ne přímo programovou chybou je nevyužití široké škály a kombinací vlivu různých instrukcí na indikátory. Program se tak zcela zbytečně nafukuje, stává se neefektivním a „přihlouplým".

*** S předchozím odstavcem přímo souvisí další typ programátorské ležérnosti. Jak běží čas, programátor si oblíbí některé instrukce na úkor jiných, až se mu „vykouří" z hlavy úplně. Jeho programy pak řeší řadu problémů oklikou, místo aby šly na věc přímo. A už vůbec nemůže vytvořit program, který by byl zajímavý a tvůrčí, protože nepracuje se všemi možnostmi, které mikroprocesor nabízí. Proto si čas od času prohlédni soupis instrukcí, jestli se tam někde „nekrčí" nějaká pozapomenutá.

*** Někdy budeš muset vymyslet krátké pomocné rutiny tak, aby byly relokovatelné - např. rutiny přenosu obrazových bitů na tiskárnu, překódování souboru znaků z různých slovních procesorů atd. Obecně se však zaměř na strukturu rutin a jejich linkováni (připojování) do zdrojového textu programu. Viz dále i stavebnicové programování.

*** Vytvářený program si rozfázuj na jednotlivé „pohyby" jako políčka filmu. Tyto části převeď do subrutin programu. Při jeho vytváření pak zjistíš, že některé subrutiny nebo jejich části můžeš sloučit - udělat z nich jednu. Kdybys program psal jako milostný román, nejen že by se ti tato řešení ani neobjevila, ale už po několika desítkách bajtů by ti tvorba programu začala „drhnout". Pamatuj - napřed struktura, pak teprve instrukce!

*** Pro programy, zvláště delší, platí jedno pravidlo - takřka neexistuje program, který by nešel zkrátit. Nebo napsat nějak jinak, třeba lépe. Proto se vždy zamysli nad tím, zda to, co jsi vymyslel, je opravdu to jediné možné nejlepší. Máš-li filipa, dříve či později

přijdeš na jiná, zajímavější řešeni. Programátor zraje spojitě s časem (naplněným prací - neplést si s hruškou na větvi!).

*** Program by měl být nejen co nejkratší, ale i co nejrychlejší (kromě časovaných rutin, kde je čas tvrdě určen). Ale ne vždy je kratší program rychlejší než jiné řešeni, které má třeba o nějaký ten bajt navíc. Sleduj, kudy všudy musí program pro splnění nějaké funkce proběhnout, a zamysli se, zda by se tato cesta nemohla odvíjet nějak jinak. Typickým případem je zasazení spousty smyček do sebe - takto vytvořená „spirálová mlhovina" může průběh operace pěkně natáhnout. Někdy je lepší takovou motanici nahradit několika inteligentními testy, které převedou řízení programu do jednoduchých kaskád, jejichž cesta je pak v součtu podstatně kratší i přehlednější.

*** Co nejdříve se nauč používat logické funkce. V nich leží ohromné možnosti krácení programu i doby jeho průběhu. Maskování není jedinou možností jejich užití. Hodně programátorů zapomíná právě na tento typ instrukcí, protože pro ně nevidí žádné užiti (... tak co se jimi zabývat?). Totéž platí o dvojkově komplementárních číslech. Je zajímavé, že špatný programátor se pozná i podle toho, že mu v programech chybějí logické operace, DK čísla a povětšinou i instrukce rotací a posuvů. Způsob užití těchto instrukcí nemusí být zrovna jen v tom, o čem se zmiňuje tato knížka - ta hovoří jen o typickém užití. Tvůrčí proces má však licenci na netušené!

*** Pokud budeš mít dojem, že ti to jde „jak psovi pastva", nic si z toho nedělej. Buď máš běžný útlum, nebo jsi se vydal zbytečně složitou cestou (struktura!). Výzkumy bylo zjištěno, že dobří programátoři laborující se strojovým kódem (assemblerem) mají produktivitu kolem 20 instrukcí denně (včetně ladění programu). Každý program zpočátku přibývá poměrně rychle. S přibývajícími bajty rychlost kynutí klesá. Někdy je dokonce nutno si říci - tak tudy ne! - a začít jinak. Neboj se přepisovat, škrtat, prostě experimentovat. Je to rozhodně lepší, než si pod sebou budovat močál a do něj pak stále hlouběji zapadat.

*** Jednou z dobrých forem práce je stavebnicové programování. Je to analogie stavebnicové korespondence, při které se za sebe řadí vhodné standardizované odstavce s případnými malými úpravami nebo doplňky. Tak se dopisy nemusejí vymýšlet stále znova. Určité rutiny, které budeš často používat, si archivuj na záznamovém médiu (v knihovně programů). Jde především o rutiny testů klávesnice, práce s obrazovou paměti (pohyby bodů na obrazovce, barevné změny), rutiny zvukového výstupu, hledání určité proměnné (i řetězcové) atd., apod. V případě potřeby patřičnou rutinu jen zařaď („nalinkuj") do programu - někdy ji ani není třeba upravovat. Nejprve si však takový archív musíš založit a vést ho tak, abys věděl, co v něm je. Tento postup je jedním ze základů úspěchu softwarových firem. Kdyby měly všechno vymýšlet stále znova, jednoduše by zkrachovaly pod tlakem časových ztrát. Proto takovou knihovnu programů vřele doporučuji. Rutiny najdeš v literatuře, ve speciálních sestavách, které jsou v prodeji na páskách a discích, některé si vytvoříš sám.

*** I když ti bude hrdost bránit, aby sis zjednodušil práci použitím Basicu, nebraň se tomu zcela. Příkazy Basicu klidně použij tam, kde by bylo programování v assembleru zbytečnou a otravnou dřinou Někdy ti Basic pomůže v „poukování" důležitých parametrů, jejichž ukládáni by jinak bylo zbytečně náročné na čas i orientaci. Rovněž nemá cenu nahrazovat strojovým kódem příkazy pro ovládání záznamu a jeho čtení. Na druhou stranu ti ale nikdo nebrání, aby ses Basicu vůbec dotýkal. Žádný assemblerový začá-

tečník nevytvoří větší program dřív než za rok. Zkus tedy postupně nahrazovat některé funkce programu v jiném jazyku rutinami strojového kódu.

*** Jednou z „horkých brambor" programování je technika, při níž program modifikuje sám sebe. Jedni - a je jich většina -ji absolutně zavrhují: „Program je nejasný, nemá čistou strukturu, je nepřevoditelný na změněné aplikační podmínky, nepromovatelný..." Jiní - těch je menšina - naopak tu a tam po této technice rádi sáhnou. Myslím, že nemá smysl tento spor řešit, protože se pohybuje spíše v rovině řečnického umění obhájců obou stran. Záleží především na tom, pro jaké účely je program konstruován, a na osobnosti programátora. Někdy se ti při programování modifikace jakoby nabídne sama. Při dalším vývoji (zdokonalování) programu většinou přijdeš na to, jak řešit věc jinak, a modifikace zase sama „odkráčí". Dokud nezvládneš základy programování, do těchto temných vod se příliš nepouštěj. Začal by ses rychle topit. Pokud jde jen o modifikaci parametrů coby dat, dá se zvládnout bez ztráty vědomí. Ale modifikace instrukcí či jejich sestav, které od adresy A dělají jedno, ale od adresy A+1 něco zcela jiného (i když na adrese A je třeba tříbajtová instrukce) a ještě se v průběhu programu modifikují, to už je silný tabák. Takový program je prakticky nerozluštitelný (i pro autora, pokud nemá perfektní dokumentaci). Jeho výhoda je v šetření paměti. A je-li opravdu geniální, i ve zvýšení prováděcí rychlosti. Protože takové programy se zpravidla nedají upravit pro změněné aplikační podmínky, není tato technika vhodná zejména v systémových a užitkových programech.

*** Podobně se při stavbě programu můžeš nechat zlákat zaváděním „umělých" testovacích příznaků do subrutin, které mohou jednou dělat to, jindy (s onou „vytestovanou" změnou) zase ono. Takovýmto sériím umělých testů a změn se snaž vyhnout, i když se ti bude zdát, že se program zbytečně „nafukuje". Rojení umělých testů vede k pozvolnému zatemňování struktury, zvyšování obtížnosti orientace v programu i snížení efektivity tvorby. Subrutiny slučuj (třeba i s použitím umělých testů) až poté, kdy ti bude program bezchybně fungovat.

*** Program je hierarchickou strukturou s rozhodovacími body v jejích nejvyšších patrech. Dbej na to, aby hustota těchto bodů byla největší na špičce celé struktury a směrem dolů prudce řídla. Binární větvení procesů v nižších patrech struktury je předem prohranou bitvou. Tam by měly být jen rutiny s jednoznačnými funkcemi volanými z vyšších pater struktury. Opačný postup spolehlivě vede k neřešitelným situacím a velmi nepříjemnému přepisování celých částí programu. Assemblerový program si na programátorovi přímo vynucuje určitou architekturu své struktury. Pokud tomuto nutkání nebudeš schopen vyhovět, nikdy žádný větší program nedokončíš.

*** Každý počítač má nějaký operační systém. Mikropočítače ho mají většinou v pamětech ROM. Sežeň si jeho údaje a komentovaný výpis. Najdeš v něm řadu podnětů, zjistíš, jak co jde dělat, a budeš moci využívat jeho rutin ve svých programech (nebudeš je muset vymýšlet a ušetříš čas i paměť svou i počítače).

*** Až zvládneš základy programování v assembleru tak, že ti nebude činit potíže tvorba jednoduchých, jednoúčelových rutin, budeš stát na prahu skutečného programování. Zásadně se nepouštěj do stavby programu bez předběžného promyšlení všech jeho funkcí. Maximální pozornost věnuj jeho komunikaci s uživatelem. Na ni do značné míry závisí užitková hodnota celého programu. Proto si budoucí program v mysli rozděl na jeho vnitřní a vnější komunikaci. Často ta vnější tvoří podstatnější část výsledného

programu. Ke tvorbě rozsáhlejšího programu přistupuj, jako bys chtěl natočit hraný film. V jedné osobě jsi scenáristou, dramaturgem i režisérem ztvárnění budoucího děje, komplikovanějšího v tom, že tvým produktem nebude „jednosměrný" filmový pás, ale struktura členitě provázaných procesů. Nakolik bude tento dynamický děj funkční i působivý ve styku s uživatelem, bude záviset na tvých schopnostech a znalostech. Tuto zcela prvotní fázi tvorby programu nikdy neopomíjej. I když ti zabere několik týdnů, neusedej k počítači, dokud tvé představy neuzrají. Nepromyšlené vyťukávání instrukcí by bylo zcela zbytečnou pantomimou. Vlastní zápis jednotlivých rutin je posledním stupněm konkretizace vytvořených představ. Bez nich se nedobereš ani obrysů základní struktury budoucího programu.

*** Když jsi při tvorbě rozsáhlejšího programu už ve fázi zápisu instrukcí do počítače, musíš se ji věnovat kontinuálně. Tato fáze klade nesmírné nároky na orientaci v celém procesu tvorby. Je přímo závislá na kvalitě průběžné interakce mezi tvou krátkodobou a dlouhodobou pamětí. Snaž se jim ulehčit vedením pomocných zápisů základních charakteristik hlavních bodů vznikající programové struktury. Tak se budeš moci koncentrovat především na tvorbu funkcí jednotlivých částí programu. Při práci na jakémkoli programovém detailu nesmíš ztratit přehled o celém dosud vytvořeném dějovém pozadí programu a v představách musíš být i kousek napřed. V podstatě jde o myšlenkovou i citovou projekci tvůrce. Pokud bys práci na programu přerušil třeba jen na kratší dobu, musel by ses s ním poté opět a namáhavě sžívat. Proto se tvorbě rozsáhlých programů prakticky nelze věnovat jako koníčku pro volné chvíle, ale buď profesionálně, nebo v podmínkách, v nichž nejsme rušeni jinými podněty. Uvedené nároky hezky shrnula jedna žena do výroku: „Po zapnutí počítače by se na obrazovce měl objevit nápis: Máš také manželku!"

*** Jedna zásada nakonec - své vědomosti stále doplňuj čtením odborné literatury a sledováním vývoje. Vše podnětné převáděj do své praxe. Opatřuj si všechnu možnou dostupnou dokumentaci o svém počítači, sháněj zajímavá programová řešení. Pokud je pro tebe hardware velkou neznámou, nezbyde ti, než se jím začít zabývat. Jinak nebudeš moci využít nic ze široké palety možností interfacingu - a to by byla ohromná škoda. Interfacing je možný už s tím, co má počítač v sobě - ovládání portů tvorby zvuku, formátu dat sériového přenosu, hrátky s obrazem, tiskárnou atd. - podle typu počítače. Tvorba a poznání tvoří nekonečnou smyčku i ve výpočetní technice.

S shakespearovským

Býti bity bit či bity býti nebit?

ti mnoho zdaru ve tvé programátorské dřině

přeje AUTOR

Příloha

Výpočet doby provedení programu **(153)**
a
diagram registrů Z80 **(154)**

Schematická znázornění průběhu vybraných instrukcí
(s uvedením parametrů
před a po provedení instrukce) **(155)**

Vývody pouzdra Z80 **(163)**
a
blokový diagram **(164)**

Instrukce Z80 zařazené do skupin **(165)**

Soubor instrukci Z80
(assemblerový tvar, hexadekadický ekvivalent
a počet taktů T) **(171)**

Vliv instrukcí Z80 na stavové indikátory **(187)**

Literatura a software **(191)**

Věcný rejstřík **(193)**

VÝPOČET DOBY PROVEDENÍ PROGRAMU

Každá instrukce se provádí určitý počet taktů T vnějších hodin. Takty jsou odvozeny od kmitočtu zabudovaného generátoru (prakticky vždy krystalového). Další časovou veličinou, k níž se váže prováděcí doba, je strojový cyklus M. Během každého cyklu M mikroprocesor provede operaci s 1 bajtem. Je pravidlem, že 1. bajt instrukce se provede za dobu 4 T, další bajty pak za dobu 3 T. Je-li tedy instrukce jednobajtové, provede se za dobu 4 T, neboli 1 M. Je-li tříbajtová, provede se za 4 + 3 + 3 = 10 T, resp 3 M Odlišnosti jsou u instrukcí, jejichž exekuce je vázána na provedení dalších interních operací (např. u instrukcí s podmínkami). Dobu provedení ovlivňují i stavy WAIT, během nichž Z80 čeká potřebný nebo stanovený počet taktů T. V praxi mohou další odlišnosti vyvstat i vlivem systému počítače (např. u Amstradu se provede každý bajt za dobu 4 T).

Budeme zvažovat počítač s hodinovým kmitočtem 2,5 MHz. Tzn., že 1 cyklus T má hodnotu 1:2,5*1000000 = 0,0000004 sec = 400 nsec. V příloze uvedené instrukce mají přiřazen počet cyklů T. Znáte—li kmitočet hodin vašeho počítače, snadno si provedete výpočet doby provedeni jakékoli části programu. Např.:

	počet cyklů T	počet provedení	doba provedení	
LD A, 35	7	1	2,8	(mikrosec.)
LD B, 49	7	1	2,8	
OR B	4	1	1.6	
AND 99	7	1	2,8	
RL A	4	1	1,6	
		celkem	11,6	
CALL XXXX	17	1	6,8	
RET	10	1	4,0	
		celkem	10,8	
LD A, 06	7	1	2,8	
LD B, 08	7	1	2,8	
LOOP:INC A	4	9	14,4	
DEC B	4	9	14,4	
JR NZ, LOOP	12 (Z = 0)	8	22,4	
	7 (Z=1)	1	4,8	
		celkem	61,6	

LD HL, 0100H	10	1	4,0
LD DE, 0200H	10	1	4,0
LD BC, 0010H	10	1	4,0
LDIR	21	15	126,0
	16) (BC = 0	1	6,4
		celkem	144,4

REGISTRY MIKROPROCESORU Z80

HLAVNI BANKA 0

A	**F**
B	**C**
D	**E**
H	**L**
IX	
IY	
SP	
PC	
I	**R**

VEDLEJŠÍ BANKA 1

A'	**F'**
B'	**C**
D'	**E'**
H'	**L'**

KLOPNÉ OBVODY PŘERUŠENÍ

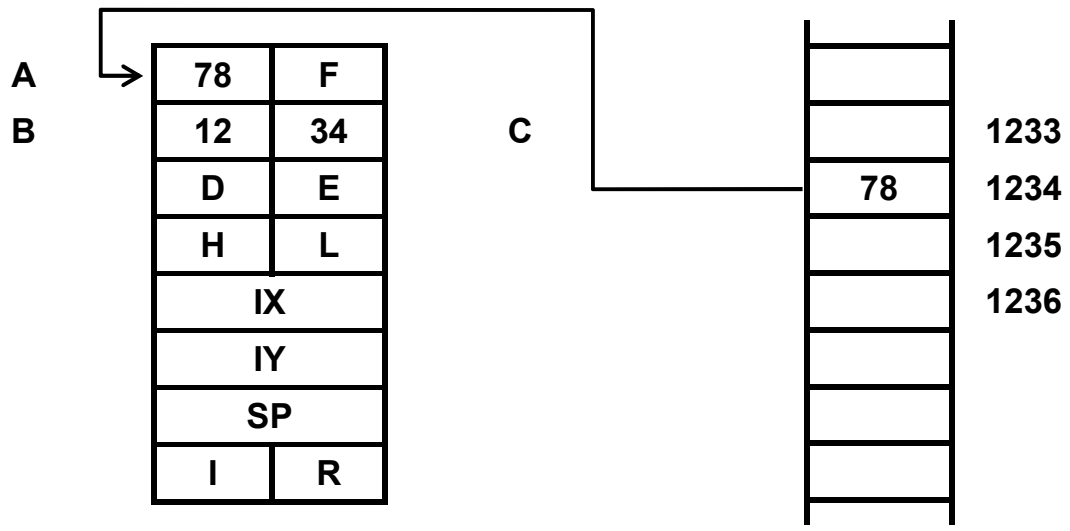
IFF1	**IFF2**	**(stav)**
IMFa	**IMFb**	**(mód)**

SCHEMATICKÁ ZNÁZORNĚNÍ

průběhu vybraných instrukcí (s uvedením parametrů před a po provedení instrukce)

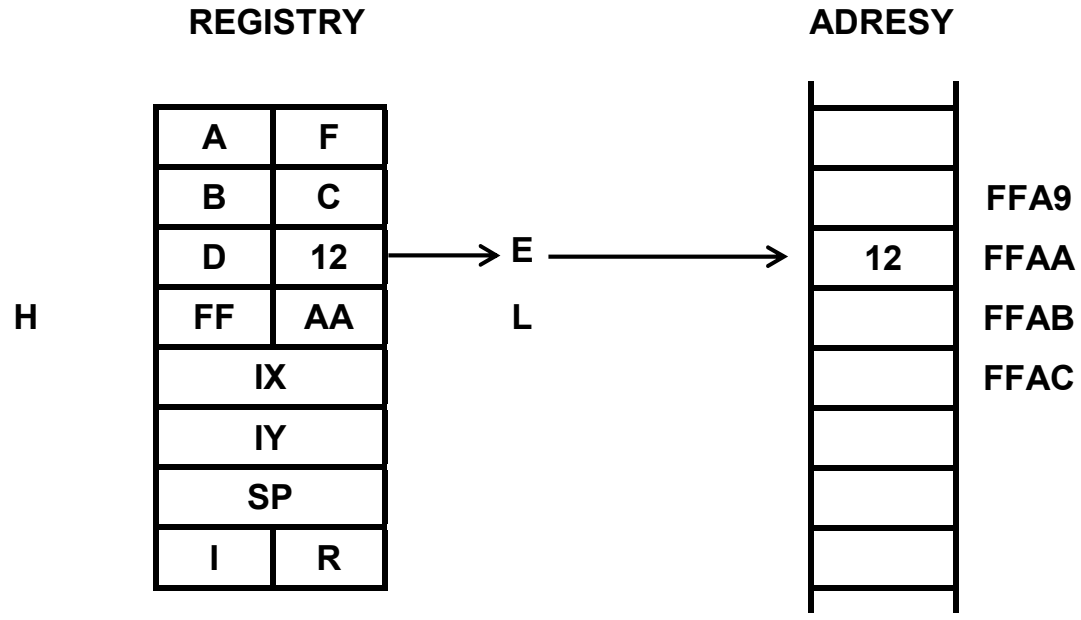
INSTRUKCE LD A, (BC)

BC =1234
(BC) = 78



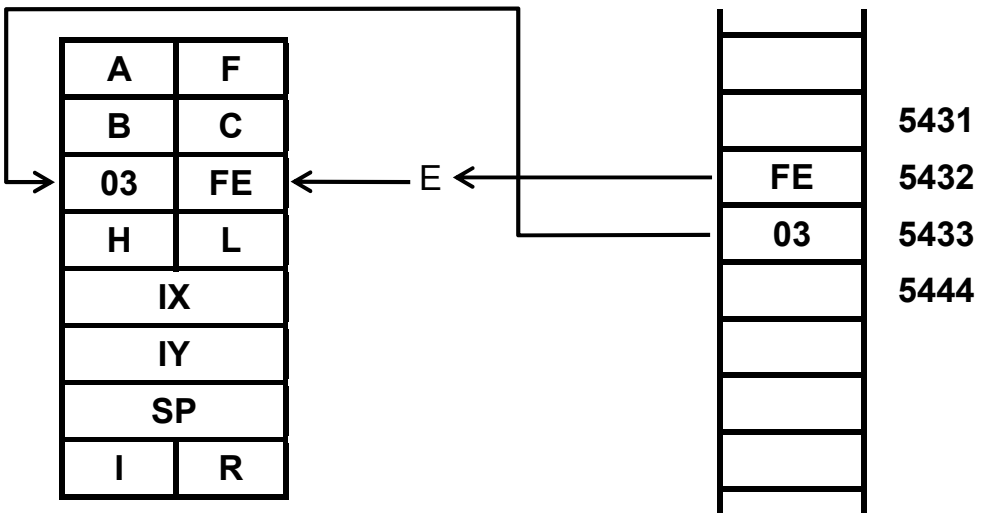
INSTRUKCE LD (HL), E

HL =FFAA
E = 12



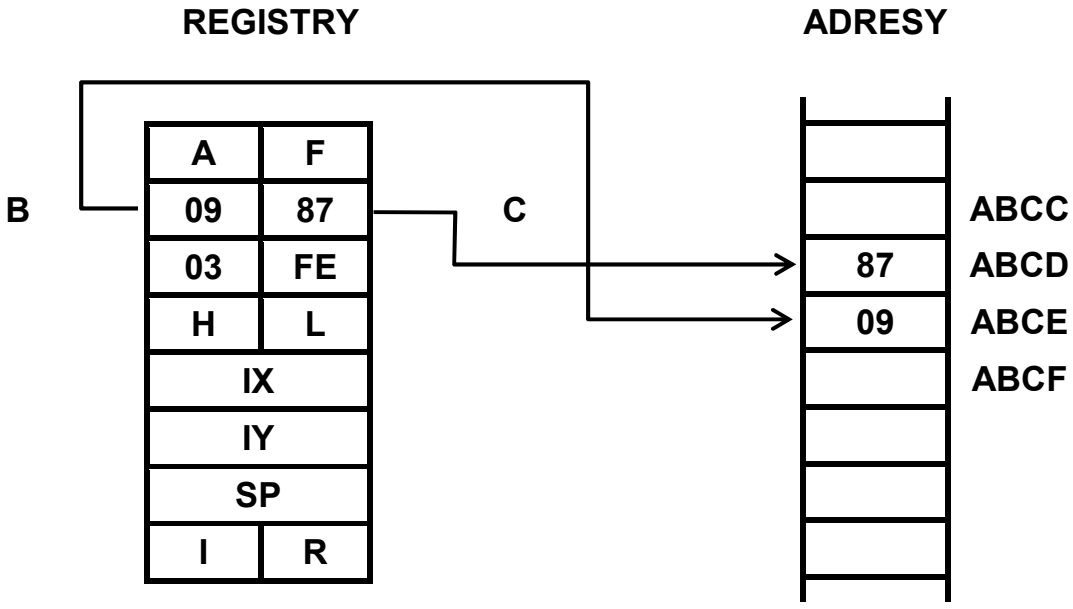
INSTRUKCE LD DE, (ADR)

ADR =5432
(ADR) = FE
(ADR+1)=03



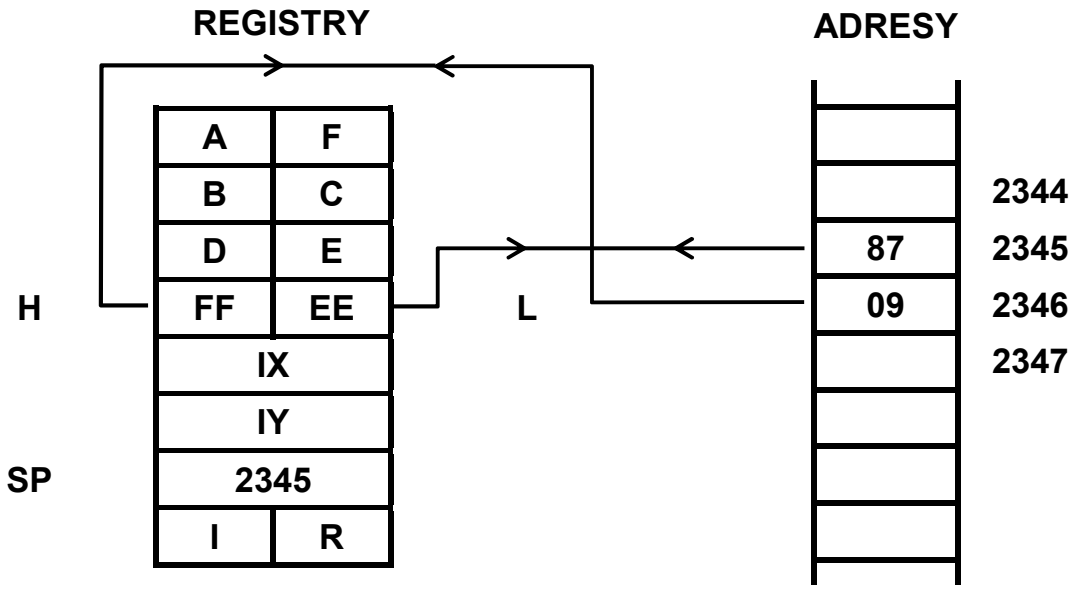
INSTRUKCE LD (ADR), BC

ADR =ABCD
BC = 0987



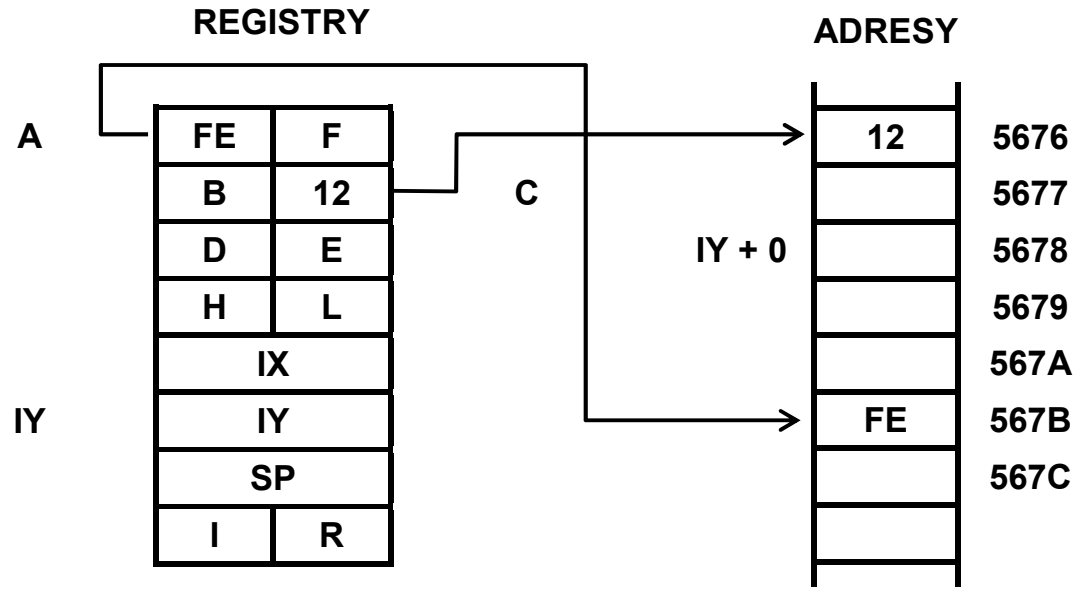
INSTRUKCE EX (SP), HL

HL = CDEF
SP = 2345
(SP) = FF
(SP+1) = EE



INSTRUKCE LD (IY + 3), A
LD (IY + FE), C

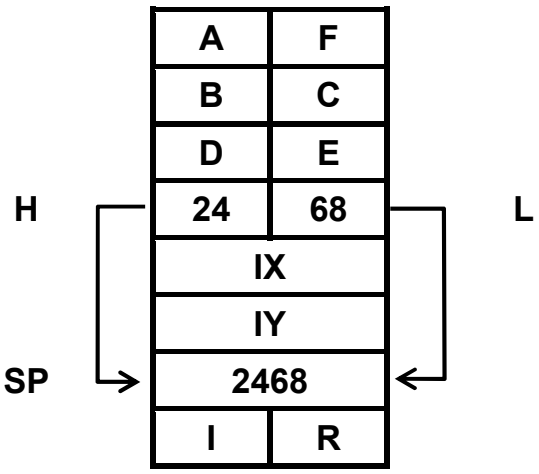
IY = 5678
A = FE
C = 12



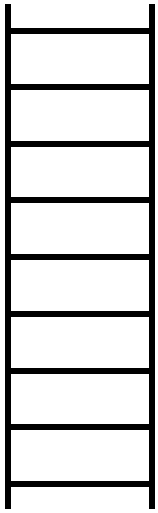
INSTRUKCE LD SP, HL

HL = 2468

REGISTRY



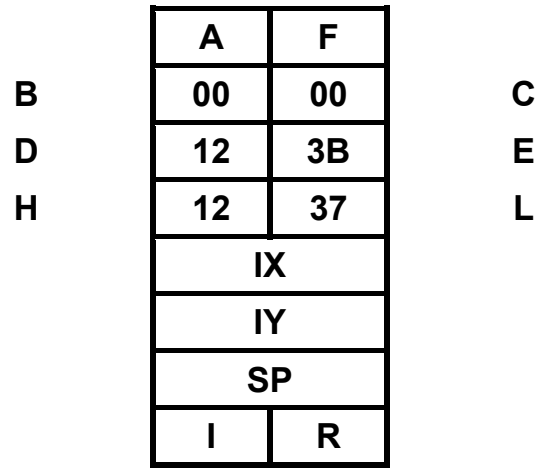
ADRESY



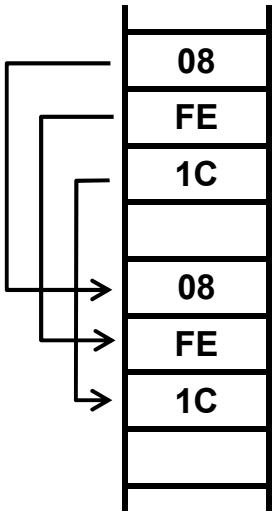
INSTRUKCE LDIR

HL = 1234
DE = 1238
BC = 0003

REGISTRY



ADRESY



INSTRUKCE CPIR

HL = 0100
BC = 0008
A = FA

REGISTRY

A	FA	F		
B	00	04	C	
	D	E		
H	01	04	L	
IX				
IY				
SP				
I		R		

ADRESY

	0100
	0101
	0102
FA	0103
	0104
	0105
	0106
	0107

INSTRUKCE CPDR

HL = 0107
BC = 0008
A = FA

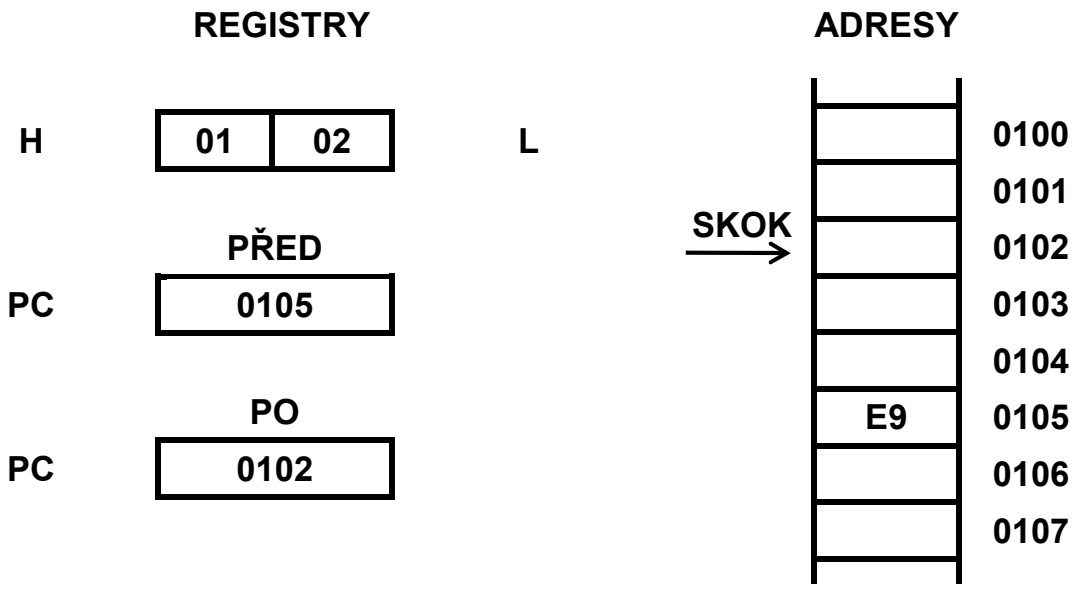
REGISTRY

A	FA	F		
B	00	03	C	
	D	E		
H	01	02	L	
IX				
IY				
SP				
I		R		

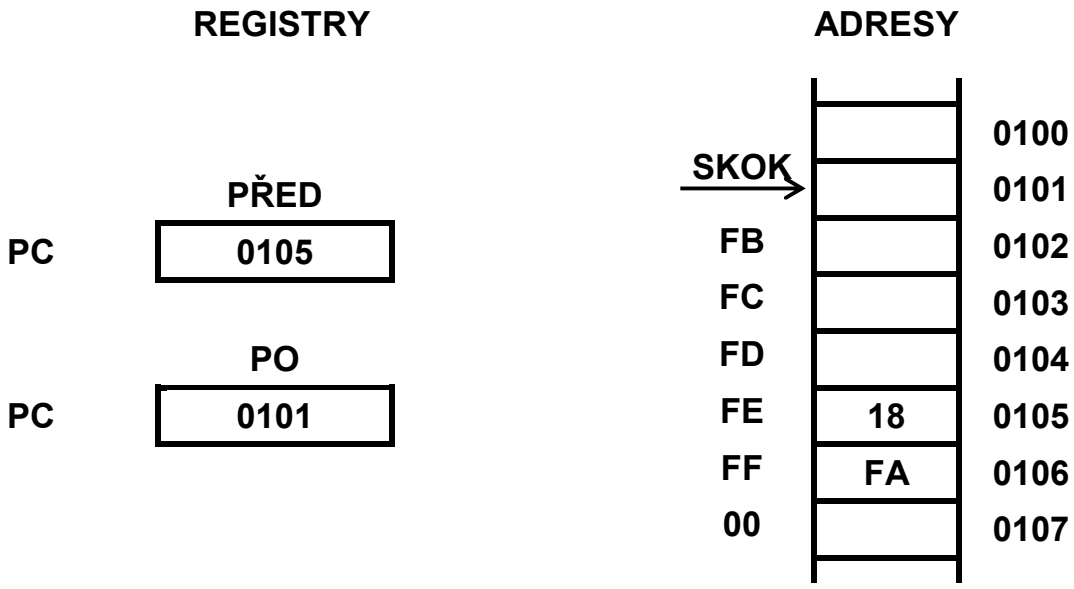
ADRESY

	0100
	0101
	0102
FA	0103
	0104
	0105
	0106
	0107

INSTRUKCE JP (HL) JE NA ADR 0105 **HL = 0102**

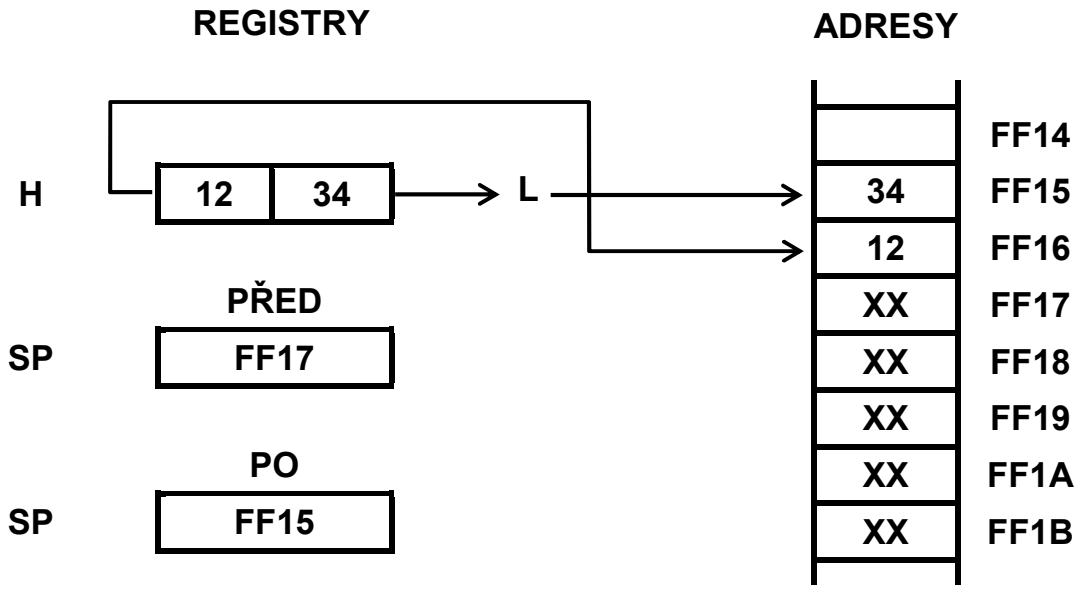


INSTRUKCE JR FA JE NA ADR 0105



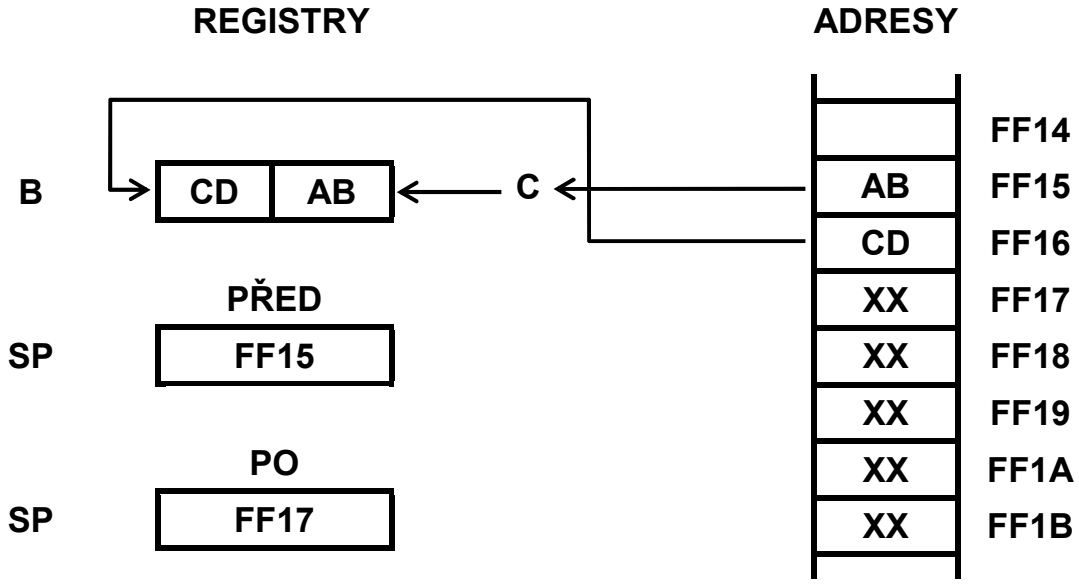
INSTRUKCE PUSH HL

HL = 1234
SP = FF17

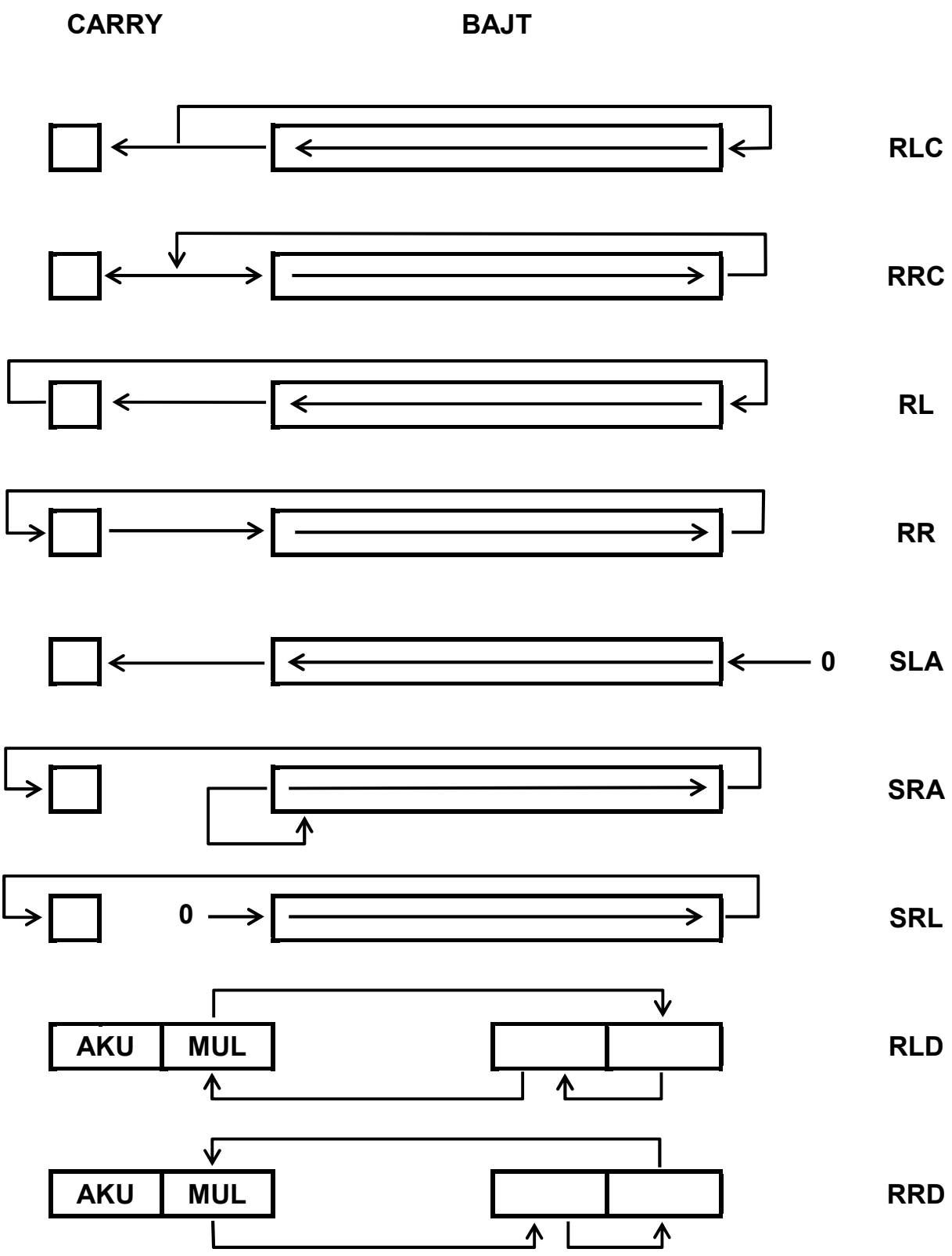


INSTRUKCE POP BCL

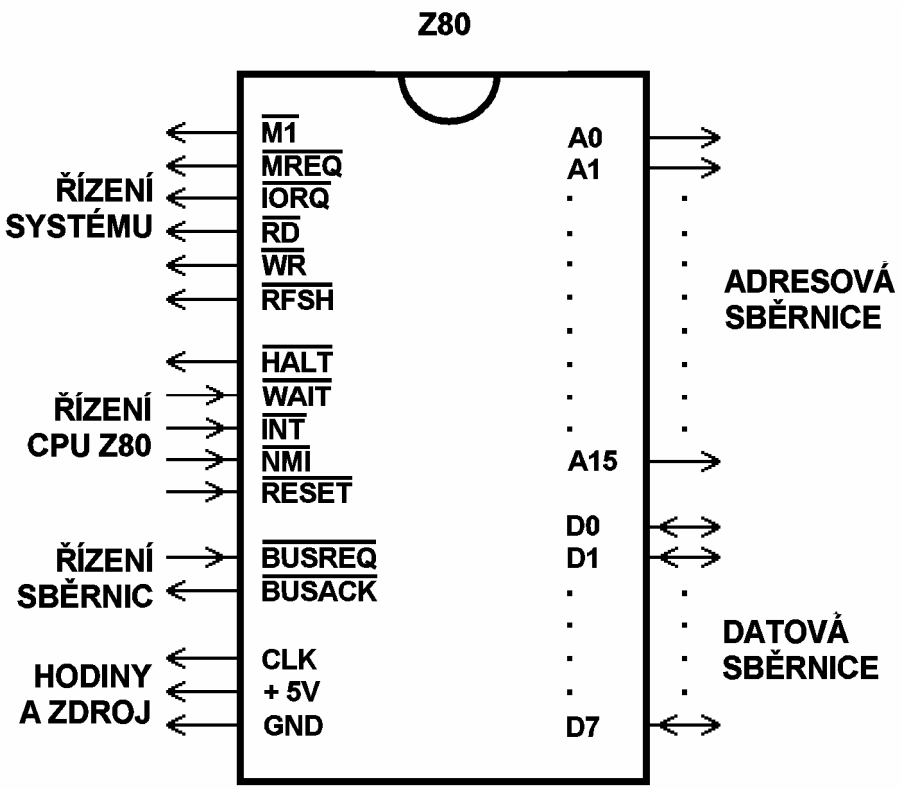
SP = FF15

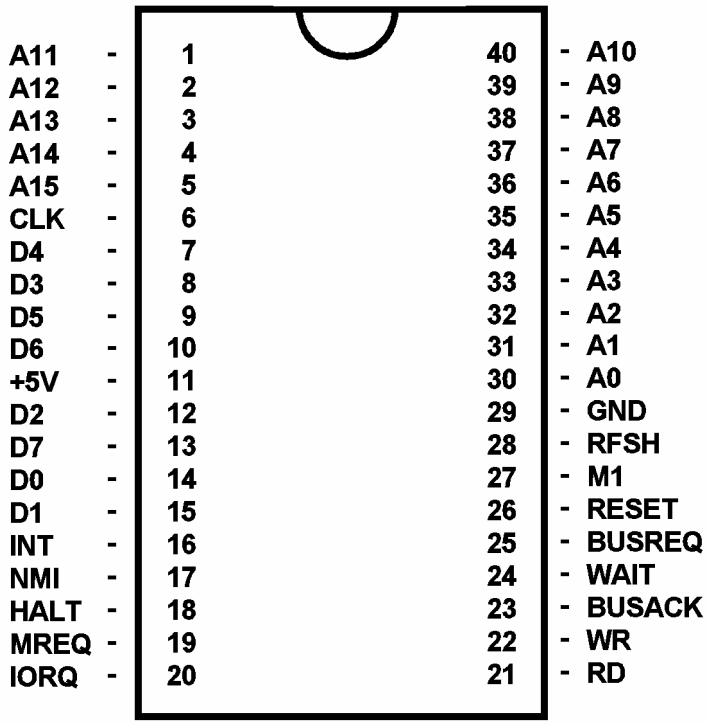


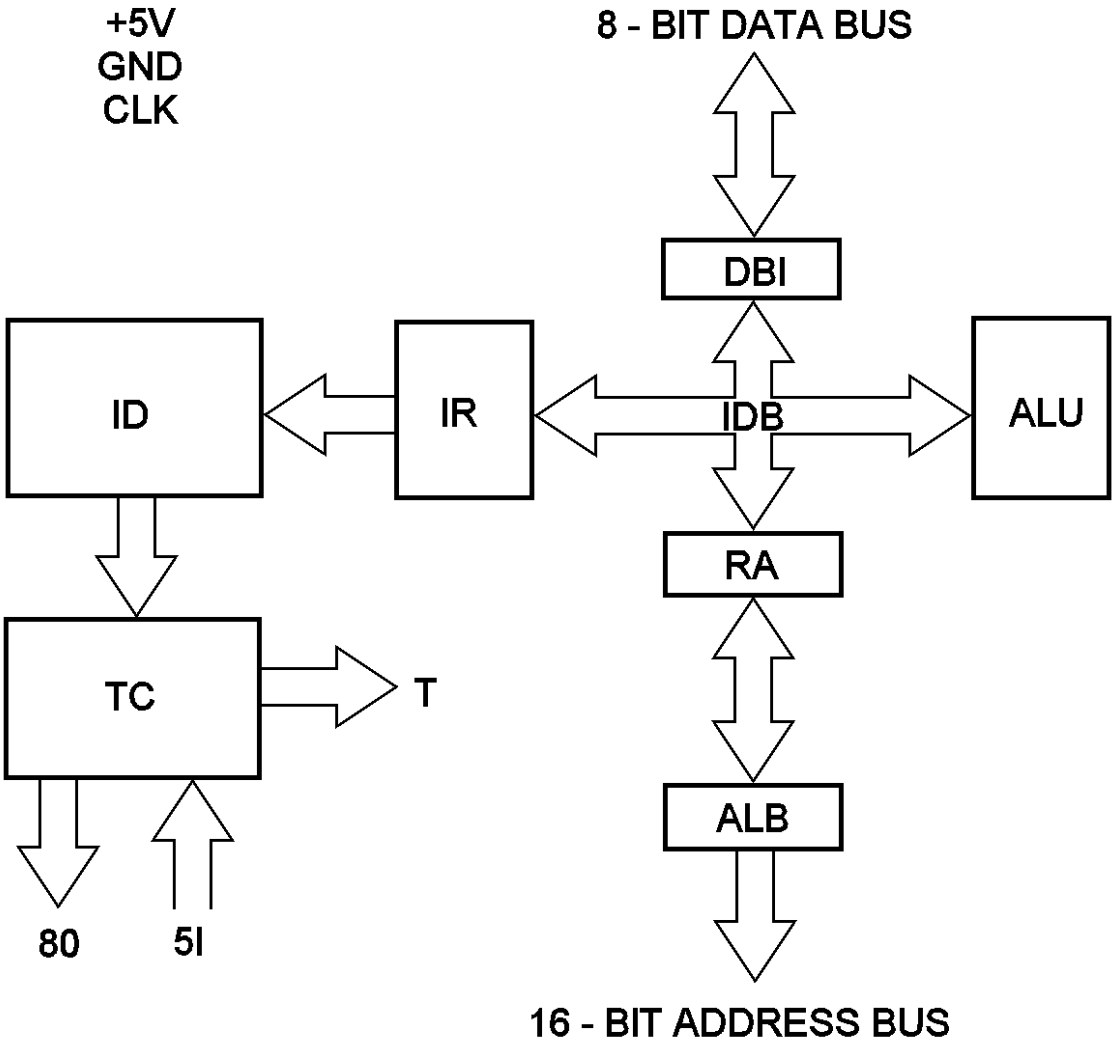
INSTRUKCE ROTACE A POSUVU **HL = 1234**



VÝVODY POUZDRA Z80 A BLOKOVÝ DIAGRAM







Anglické symboly a jejich překlad.

8-BIT DATA BUS - 8-bitova datová sběrnice
16-BIT ADDRESS BUS - 16-bitová adresová sběrnice
T - CPU TIMING - časování mikroprocesoru
80-8 SYSTEMS AND CPU CONTROL OUTPUTS - 8 výstupů pro řízeni systémů a mikroprocesoru
5I - 5 CPU CONTROL INPUTS - 5 vstupů pro řízeni mikroprocesoru
DBI - DATA BUS INTERFACE - interface datové sběrnice
IDB - INTERNAL DATA BUS - vnitřní datová sběrnice
ALU - ARITHMETIC AND LOGIC UNIT - aritmetická a logická jednotka
IR - INSTRUCTION REGISTER - registr instrukce
ID - INSTRUCTION DECODER - dekodér instrukce
TC - CPU TIMING CONTROL - řízeni časování mikroprocesoru
RA - REGISTER ARRAY - maticový registr
ALB - ADDRESS LOGIC AND BUFFERS - adresovací logika a buffery

SKUPINY INSTRUKCÍ

8-BITOVÝ PŘENOS

	B	C	D	E	H	L	(HL)	A	(BC)	(DE)	(ADR)	NN
LD B, .	40	41	42	43	44	45	46	47				06XX
LD C, .	48	49	4A	4B	4C	4D	4E	4F				0EXX
LD D, .	50	51	52	53	54	55	56	57				16XX
LD E, .	58	59	5A	5B	5C	5D	5E	5F				1 EXX
LD H, .	60	61	62	63	64	65	66	67				26XX
LD L, .	68	69	6A	6B	6C	60	6E	6F				2EXX
LD (HL), .	70	71	72	73	74	75	76	77				36XX
LD A, .	78	79	7A	7B	7C	7D	7E	7F	0A	1A	3AXXXX	3EXX
LD (BC), .	02											
LO(DE), .	12											
LD (ADR), .	32XXXX											

	B	C	D	E	H	L	A
LD ., (IX+d)	0046XX	D04EXX	DD56XX	DD5EXX	DD66XX	DD6EXX	DD7EXX
LD .,(IY+d)	FD46XX	F04EXX	FD56XX	FD5EXX	FD66XX	FD6EXX	FD7EXX
LD (IX + d), .	DD70XX	DD71XX	DD72XX	DD73XX	DD74XX	DD75XX	DD77XX
LD (IY+d), .	FD70XX	FD71XX	FD72XX	FD73XX	FD74XX	FD75XX	FD77XX

LD |IX + d), NN	DD36XXNN
LD (IY + d), NN	FD36XXNN
LD I, A	ED47
LD R,A	ED4F
LD A, I	ED57
LD A, R	ED5F

16-BITOVÝ PŘENOS

	BC	DE	HL	SP	IX	IY
LD ., NNNN	01NNNN	11NNNN	21NNNN	31NNNN	DD21NNNN	FD21NNNN
LD ., (ADR)	ED4BXXXX	ED5BXXXX	2AXXXX	ED7BXXXX	DD2AXXXX	FD2AXXXX
LD (ADR), .	ED43XXXX	ED53XXXX	22XXXX	ED73XXXX	DD22XXXX	FD22XXXX
LD SP, .			F9		DDF9	FDF9

	BC	DE	HL	AF	IX	IY
POP .	C1	D1	E1	F1	ODE1	FDE1
PUSH .	C5	D5	E5	F5	DDE5	FDE5

EX AF, AF'	08
EXX	D9
EX DE, HL	EB
EX(SP),HL	E3
EX(SP),IX	DDE3
EX(SP),IY	FDE3

BLOKOVÝ PŘENOS A PROHLEDÁVÁNÍ

LDI	EDA0	CPI	EDA1
LDD	EDA	CPD	EDA9
LDIR	EDB0	CPIR	EDB1
LDDR	EDB8	CPDR	EDB9

8-BITOVÁ ARITMETIKA A LOGIKA

	B	C	D	E	H	L	(HL)	A	NN	(IX + d)	(IY + d)
ADD A,	80	81	82	83	84	85	86	87	C6XX	DD86XX	FD86XX
ADC A,	88	89	8A	8B	8C	8D	8E	8F	CEXX	DD8EXX	FD8EXX
SUB	90	91	92	93	94	95	96	97	D6XX	0D96XX	FD96XX
SBC A,	98	99	9A	9B	9C	9D	9E	9F	DEXX	DD9EXX	FD9EXX
AND	A0	A1	A2	A3	A4	A5	A6	A7	E6XX	DDA6XX	FDA6XX
XOR	A8	A9	AA	AB	AC	AD	AE	AF	EEXX	DDAEXX	FDAEXX
OR	B0	B1	B2	B3	B4	B5	B6	B7	F6XX	DDB6XX	FDB6XX
CP	B8	B9	BA	BB	BC	BD	BE	BF	FEXX	DDBEXX	FDBEXX
INC	C4	0C	14	1C	24	2C	34	3C		DD34XX	F034XX
DEC	05	0D	15	1D	25	2D	35	3D		DD35XX	FD35XX
DAA	27										
CPL	2F										
NEG	ED44										

16-BITOVÁ ARITMETIKA

	BC	DE	HL	SP	IX	IY
INC	03	13	23	33	DD23	FD23
DEC	0B	1B	2B	3B	DD2B	FD2B
ADD HL,	09	19	29	39		
ADC HL,	ED4A	ED5A	ED6A	ED7A		
SBC HL,	ED42	ED52	ED62	ED72		
ADD IX,	DD09	DD19		D039	DD29	
ADD IY,	FD09	FD19		FD39		FD29

ROTACE A POSUVY

	B	C	D	E	H	L	(HL)	A	(IX + d)	(IY + d)
RLC .	CB00	CB01	CB02	CB03	CB04	CB05	CB06	CB07	DDCBXX06	FDCBXX06
RRC .	CB08	CB09	CB0A	CB0B	CB0C	CB0D	CB0E	CB0F	DDCBXX0E	FDCBXX0E
RL .	CB10	CB11	CB12	CB13	CB14	CB15	CB16	CB17	DDCBXX16	FDCBXX16
RR .	CB18	CB19	CB1A	CB1B	CB1C	CB1D	CB1E	CB1F	DDCBXX1E	F0CBXX1E
SLA .	CB20	CB21	CB22	CB23	CB24	CB25	CB26	CB27	DDCBXX26	FDCBXX26
SRA .	CB28	CB29	CB2A	CB2B	CB2C	CB2D	CB2E	CB2F	DDCBXX2E	FDCBXX2E
SLL .	CB30	CB31	CB32	CB33	CB34	CB35	CB36	CB37	DDCBXX36	FDCBXX36
SRL .	CB38	CB39	CB3A	CB3B	CB3C	CB3D	CB3E	CB3F	DDCBXX3E	FDCBXX3E

RLCA 07
RRCA 0F
RLA 17
RRA 1F
RRD ED67
RLD ED6F

BITOVÉ MANIPULACE

	B	C	D	E	H	L	(HL)	A	(IX + d)	(IY + d)
BIT 0	CB40	CB41	CB42	CB43	CB44	CB45	CB46	CB47	DDCBXX46	FDCBXX46
BIT 1	CB48	CB49	CB4A	CB4B	CB4C	CB4D	CB4E	CB4F	DDCBXX4E	FDCBXX4E
BIT 2	CB50	CB51	CB52	CB53	CB54	CB55	CB56	CB57	DDCBXX56	FDCBXX56
BIT 3	CB58	CB59	CB5A	CB5B	CB5C	CB5D	CB5E	CB5F	00CBXX5E	FDCBXX5E
BIT 4	B60	CB61	CB62	CB63	CB64	CB65	CB66	CB67	DDCBXX66	FDCBXX66
BIT 5	CB68	CB69	CB6A	CB6B	CB6C	CB6D	CB6E	CB6F	DDCBXX6E	FDCBXX6E
BIT 6	CB70	CB71	CB72	CB73	CB74	CB75	CB76	CB77	DDCBXX76	FDCBXX76
BIT 7	CB78	CB79	CB7A	CB7B	CB7C	CB7D	CB7E	CB7F	DDCBXX7E	FDCBXX7E
RES 0	CB80	CB81	CB82	CB83	CB84	CB85	CB86	CB87	00CBXX86	FDCBXX86
RES 1	CB88	CB89	CB8A	CB8B	CB8C	CB8D	CB8E	CB8F	DDCBXX8E	FDCBXX8E
RES 2	CB90	CB91	CB92	CB93	CB94	CB95	CB96	CB97	DDCBXX96	FDCBXX96
RES 3	CB98	CB99	CB9A	CB9B	CB9C	CB9D	CB9E	CB9F	DDCBXX9E	FDCBXX9E
RES 4	CBA0	CBA1	CBA2	CBA3	CBA4	CBA5	CBA6	CBA7	DDCBXXA6	FDCBXXA6
RES 5	CBA8	CBA9	CBAA	CBAB	CBAC	CBAD	CBAE	CBAF	DDCBXXAE	FDCBXXAE
RES 6	CBB0	CBB1	CBB2	CBB3	CBB4	CBB5	CBB6	CBB7	DDCBXXB6	FDCBXXB6
RES 7	CBB8	CBB9	CBBA	CBBB	CBBC	CBBD	CBBE	CBBF	DDCBXXBE	FDCBXXBE
RES 0	CBC0	CBC1	CBC2	CBC3	CBC4	CBC5	CBC6	CBC7	DDCBXXC6	FDCBXXC6
RES 1	CBC8	CBC9	CBCA	CBCB	CBCC	CBCD	CBCE	CBCF	DDCBXXCE	FDCBXXCE
RES 2	CBD0	CBD1	CBD2	CBD3	CBD4	CBD5	CBD6	CBD7	DDCBXXD6	FDCBXXD6
RES 3	CBD8	CBD9	CBDA	CBDB	CBDC	CBDD	CBDE	CBDF	DDCBXXDE	FDCBXXDE
RES 4	CBE0	CBE1	CBE2	CBE3	CBE4	CBE5	CBE6	CBE7	DDCBXXE6	FDCBXXE6
RES 5	CBE8	CBE9	CBEA	CBEB	CBEC	CBED	CBEE	CBEF	DDCBXXEE	FDCBXXEE
RES 6	CBF0	CBF1	CBF2	CBF3	CBF4	CBF5	CBF6	CBF7	0DCBXXF6	F0CBXXF6
RES 7	CBF8	CBF9	CBFA	CBFB	CBFC	CBFD	CBFE	CBFF	DDCBXXFE	FDCBXXFE

SKOKY, VOLÁNÍ A NÁVRATY

		(HL)	(IX)	(IY)
JR	18XX			
JP	C3XXXX	E9	DDE9	FDE9
CALL	CDXXXX			
RET	C9			

podm:	NZ	Z	NC	C	PO	PE	P	M
JR	20XX	28XX	38XX	38XX				
JP	C2XXXX	CAXXXX	D2XXXX	DAXXXX	E2XXX X	EAXXXX	F2XXXX	FAXXXX
CALL	C4XXXX	CCXXXX	D4XXXX	DCXXXX	E4XXX X	ECXXXX	F4XXXX	FCXXXX
RET	C0	C8	D0	D8	E0	E8	F0	F8

DJNZ	10XX
RETN	ED45
RETI	ED4D

VSTUP/VÝSTUP

	B	C	D	E	H	L	A
IN ., (C)	ED40	ED48	ED50	ED58	ED60	ED68	ED78
OUT (C), .	ED41	ED49	ED51	ED59	ED61	ED69	ED79
IN A, (port)	DBXX						
OUT (port), A	D3XX						

INI	EDA2	OUTI	EDA3
IND	EDAA	OUTD	EDAB
INIR	EDB2	OTIR	EDB3
INDR	EDBA	OTDR	EDBB

OSTATNÍ INSTRUKCE (ŘÍZENÍ MIKROPROCESORU)

NOP	00				
SCF	37	CCF	3F		
HALT	76				
DI	F3	EI	FB		
IM 0	ED46	IM 1	ED56	IM2	EQ5E

Následující stránky s přehledem instrukcí jsou záměrně potištěny jednostranně, aby si je čtenář mohl opatrně vyříznout a sestavit podle své potřeby.

INSTRUKCE CPU Z80

ADC A, (HL)	8E	7
ADC A, (IX+ dis)	DD 8E XX	19
ADC A, (IY+ dis)	FD 8E XX	19
ADC A, A	8F	4
ADC A, B	88	4
ADC A, C	89	4
ADC A, D	8A	4
ADC A, E	8B	4
ADC A, H	8C	4
ADC A, L	8D	4
ADC A, NN	CE NN	7
ADC HL, BC	ED 4A	15
ADC HL, DE	ED 5A	15
ADC HL, HL	ED 6A	15
ADC HL, SP	ED 7A	15
ADD A, (HL)	86	7
ADD A, (IX + dis)	DD 86 XX	19
ADD A, (IY + dis)	FD 86 XX	19
ADD A, A	87	4
ADD A, B	80	4
ADD A, C	81	4
ADD A, D	82	4
ADD A, E	83	4
ADD A, H	84	4
ADD A, L	85	4
ADD A, NN	C6 NN	7
ADD HL, BC	09	11
ADD HL, DE	19	11
ADD HL, HL	29	11
ADD HL, SP	39	11
ADD IX, BC	DD 09	15
ADD IX, DE	DD 19	15
ADD IX, IX	DD 29	15
ADD IX, SP	DD 39	15
ADD IY, BC	FD 09	15
ADD IY, DE	FD 19	15
ADD IY, IY	FD 29	15
ADD IY, SP	FD 39	15
AND (HL)	A6	7
AND (IX + dis)	DD A6 XX	19
AND (IY + dis)	FD A6 XX	19
AND A	A7	4
AND B	A0	4
AND C	A1	4
AND D	A2	4
AND E	A3	4
AND H	A4	4
AND L	A5	4
AND NN	E6 NN	7
BIT 0, (HL)	CB 46	12
BIT 0, (IX + dis)	DD CB XX 46	20
BIT 0, (IY + dis)	FD CB XX 46	20
BIT 0, A	CB 47	8
BIT 0, B	CB 40	8
BIT 0, C	CB 41	8
BIT 0, D	CB 42	8
BIT 0, E	CB 43	8
BIT 0, H	CB 44	8
BIT 0, L	CB 45	8
BIT 1, (HL)	CB 4E	12
BIT 1, (IX + dis)	DD CB XX 4E	20
BIT 1, (IY + dis)	FD CB XX 4E	20
BIT 1, A	CB 4F	8
BIT 1, B	CB 48	8
BIT 1, C	CB 49	8
BIT 1, D	CB 4A	8
BIT 1, E	CB 4B	8
BIT 1, H	CB 4C	8
BIT 1, L	CB 4D	8
BIT 2, (HL)	CB 56	12
BIT 2, (IX + dis)	DD CB XX 56	20
BIT 2, (IY + dis)	FD CB XX 56	20
BIT 2, A	CB 57	8
BIT 2, B	CB 50	8
BIT 2, C	CB 51	8
BIT 2, D	CB 52	8
BIT 2, E	CB 53	8
BIT 2, H	CB 54	8
BIT 2, L	CB 55	8
BIT 3, (HL)	CB 5E	12

BIT 3, (IX + dis)	DD CB XX 5E	20
BIT 3, (IY + dis)	FD CB XX 5E	20
BIT 3, A	CB 5F	8
BIT 3, B	CB 58	8
BIT 3, C	CB 59	8
BIT 3, D	CB 5A	8
BIT 3, E	CB 5B	8
BIT 3, H	CB 5C	8
BIT 3, L	CB 5D	8
BIT 4, (HL)	CB 66	12
BIT 4, (IX + dis)	DD CB XX 66	20
BIT 4, (IY + dis)	FD CB XX 66	20
BIT 4, A	CB 67	8
BIT 4, B	CB 60	8
BIT 4, C	CB 61	8
BIT 4, 0	CB 62	8
BIT 4, E	CB 63	8
BIT 4, H	CB 64	8
BIT 4, L	CB 65	8
BIT 5, (HL)	CB 6E	12
BIT 5, (IX+ dis)	DD CB XX 6E	20
BIT 5, (IY+ dis)	FD CB XX 6E	20
BIT 5, A	CB 6F	8
BIT 5, B	CB 68	8
BIT 5, C	CB 69	8
BIT 5, D	CB 6A	8
BIT 5, E	CB 6B	8
BIT 5, H	CB 6C	8
BIT 5, L	CB 6D	8
BIT 6, (HL)	CB 76	12
BIT 6, (IX+ dis)	DD CB XX 76	20
BIT 6, (IY + dis)	FD CB XX 76	20
BIT 6, A	CB 77	8
BIT 6, B	CB 70	8
BIT 6, C	CB 71	8
BIT 6, D	CB 72	8
BIT 6, E	CB 73	8
BIT 6, H	CB 74	8
BIT 6, L	CB 75	8
BIT 7, (HL)	CB 7E	12
BIT 7, (IX + dis)	DD CB XX 7E	20
BIT 7, (IY + dis)	FD CB XX 7E	20
BIT 7, A	CB 7F	8
BIT 7, B	CB 78	8
BIT 7, C	CB 79	8
BIT 7, D	CB 7A	8
BIT 7, E	CB 7B	8
BIT 7, H	CB 7C	8
BIT 7, L	CB 7D	8
CALL ADDR	CD XX XX	17
CALL Z, ADDR	CC XX XX	17/10
CALL NZ, ADDR	C4 XX XX	17/10
CALL C, ADDR	DC XX XX	17/10
CALL NC, ADDR	D4 XX XX	17/10
CALL PE, ADDR	EC XX XX	17/10
CALL PO, ADDR	E4 XX XX	17/10
CALL H, ADDR	FC XX XX	17/10
CALL P, ADDR	F4 XX XX	17/10
CCF	3F	4
CP (HL)	BE	7
CP (IX + dis)	DD BE XX	19
CP (IY + dis)	FD BE XX	19
CP A	BF	4
CP B	B8	4
CP C	B9	4
CP D	BA	4
CP E	BB	4
CP H	BC	4
CP L	BD	4
CP NN	FE NN	7
CP D	ED A9	16
CPDR	ED B9	21/16
CPI	ED A1	16
CPIR	ED B1	21/16
CPL	2F	4
DAA	27	4
DEC (HL)	35	11
DEC (IX + dis)	DD 35 XX	23
DEC (IY + dis)	FD 35 XX	23
DEC A	3D	4
DEC B	05	4
DEC C	0D	4
DEC D	15	4
DEC E	1D	4
DEC H	25	4
DEC L	2D	4
DEC BC	0B	6
DEC DE	1B	6
DEC HL	2B	6
DEC IX	DD 2B	10

DEC IY	FD 2B	10
DEC SP	3B	6
DI	F3	4
DJNZ, dis	10 XX	13/8
EI	FB	4
EX (SP), HL	E3	19
EX (SP), IX	DD E3	23
EX (SP), IY	FD E3	23
EX AF, AF'	08	4
EX DE, HL	EB	4
EXX	D9	4
HALT	76	4
IM 0	ED 46	8
IM 1	ED 56	8
IM 2	ED 5E	8
IN A, (port)	DB NN	11
IN A, (C)	ED 78	12
IN B, (C)	ED 40	12
IN C, (C)	ED 48	12
IN D, (C)	ED 50	12
IN E, (C)	ED 58	12
IN H, (C)	ED 60	12
IN L, (C)	ED 68	12
INC (HL)	34	11
INC (IX + dis)	DO 34 XX	23
INC (IY + dis)	FD 34 XX	23
INC A	3C	4
INC B	04	4
INC C	0C	4
INC D	14	4
INC E	1C	4
INC H	24	4
INC L	2C	4
INC BC	03	6
INC DE	13	6
INC HL	23	6
INC IX	DD 23	10
INC IY	FD 23	10
INC SP	33	6
IND	ED AA	16
INI	ED A2	16
INDR	ED BA	21/16
INIR	ED B2	21/16
JP (HL)	E9	4
JP (IX)	DD E9	8
JP (IY)	FD E9	8
JP ADDR	C3 XX XX	10
JP Z, ADDR	CA XX XX	10
JP NZ, ADDR	C2 XX XX	10
JP C, ADDR	DA XX XX	10
JP NC, ADDR	D2 XX XX	10
JP PE, ADDR	EA XX XX	10
JP PO, ADDR	E2 XX XX	10
JP M, ADDR	FA XX XX	10
JP P, ADDR	F2 XX XX	10
JR dis	18 XX	12
JR Z, dis	28 XX	12/7
JR NZ, dis	20 XX	12/7
JR C, dis	38 XX	12/7
JR NC, dis	30 XX	12/7
LD (ADDR), A	32 XX XX	13
LD (ADDR), BC	ED 43 XX XX	20
LD (ADDR),DE	ED 53 XX XX	20
LD (ADDR), HL	ED 63 XX XX	20
LD (ADDR),HL	22 XX XX	16
LD (ADDR), IX	DD 22 XX XX	20
LD (ADDR), IY	FD 22 XX XX	20
LD (ADDR), SP	ED 73 XX XX	20
LD (BC),A	02	7
LD (DE),A	12	7
LD (HL),A	77	7
LD (HL), B	70	7
LD (HL),C	71	7
LD (HL),D	72	7
LD (HL),E	73	7
LD (HL),H	74	7
LD (HL), L	75	7
LD (HL), NN	36 NN	10
LD (IX + dis), A	DD 77 XX	19
LD (IX + dis), B	DD 70 XX	19
LD (IX + dis), C	DD 71 XX	19
LD (IX + dis), D	DD 72 XX	19
LD (IX + dis), E	DO 73 XX	19
LD (IX + dis),H	DD 74 XX	19
LD (IX + dis), L	DD 75 XX	19
LD (IX + dis), NN	DD 36 XX NN	19
LD (IY + dis), A	FD 77 XX	19
LD (IY + dis), B	FD 70 XX	19
LD (IY + dis), C	FD 71 XX	19
LD (IY + dis), D	FD 72 XX	19

LD (IY + dis), E	FD 73 XX	19
LD (IY + dis), H	FD 74 XX	19
LD (IY + dis), L	FD 75 XX	19
LD (IY + dis), NN	FD 36 XX NN	19
LD A, (ADDR)	3A XX XX	13
LD A, (BC)	0A	7
LD A, (DE)	1A	7
LD A, (HL)	7E	7
LD A, (IX + dis)	DD 7E XX	19
LD A, (IY + dis)	FD 7E XX	19
LD A, A	7F	4
LD A, B	78	4
LD A, C	79	4
LD A, D	7A	4
LD A, E	7B	4
LD A, H	7C	4
LD A, L	7D	4
LD A, NN	3E NN	7
LD A, I	ED 57	9
LD A, R	ED 5F	9
LD B, (HL)	46	7
LD B, (IX + dis)	DD 46 XX	19
LD B, (IY + dis)	FD 46 XX	19
LD B, A	47	4
LD B, B	40	4
LD B, C	41	4
LD B, D	42	4
LD B, E	43	4
LD B, H	44	4
LD B, L	45	4
LD B, NN	06 NN	7
LD BC, (ADDR)	ED 4B XX XX	20
LD BC, NNNN	01 NN NN	10
LD C, (HL)	4E	7
LD C, (IX + dis)	DD 4E XX	19
LD C, (IY + dis)	FD 4E XX	19
LD C, A	4F	4
LD C, B	48	4
LD C, C	49	4
LD C, D	4A	4
LD C, E	48	4
LD C, H	4C	4
LD C, L	4D	4
LD C, NN	0E NN	7
LD D, (HL)	56	7
LD D, (IX + dis)	DD 56 XX	19
LD D, (IY + dis)	FD 56 XX	19
LD D, A	57	4
LD D, B	50	4
LD D, C	51	4
LD D, D	52	4
LD D, E	53	4
LD D, H	54	4
LD D, L	55	4
LD D, NN	16 NN	7
LD DE, (ADDR)	ED 5B XX XX	20
LD DE, NNNN	11 NNNN	10
LD E, (HL)	5E	7
LD E, (IX + dis)	DD 5E XX	19
LD E, (IY + dis)	FD 5E XX	19
LD E,A	5F	4
LD E, B	58	4
LD E, C	59	4
LD E, D	5A	4
LD E, E	5B	4
LD E, H	5C	4
LD E, L	5D	4
LD E, NN	1E NN	7
LD H, (HL)	66	7
LD H, (IX + dis)	DD 66 XX	19
LD H, (IY + dis)	FD 66 XX	19
LD H, A	67	4
LD H, B	60	4
LD H, C	61	4
LD H, D	62	4
LD H, E	63	4
LD H, H	64	4
LD H, L	65	4
LD H, NN	26 NN	7
LD HL, (ADDR)	ED 6B XX XX	20
LD HL, (ADDR)	2A XX XX	16
LD HL, NNNN	21 NN NN	10
LD L, (HL)	6E	7
LD L, (IX + dis)	DD 6E XX	19
LD L, (IY + dis)	FD 6E XX	19
LD L, A	6F	4
LD L, B	68	4
LD L, C	69	4
LD L, D	6A	4
LD L, E	6B	4

LD L,H	6C	4
LD L, L	6D	4
LD L, NN	2E NN	7
LD IX, (ADDR)	DD 2A XX XX	20
LD IX, NNNN	DD 21 NN NN	14
LD IY, (ADDR)	FD 2A XX XX	20
LD IY, NNNN	FD 21 NN NN	14
LD I, A	ED 47	9
LD R, A	ED 4F	9
LD SP, (ADDR)	ED 7B XX XX	20
LD SP, HL	F9	6
LD SP, IX	DD F9	10
LD SP, IY	FD F9	10
LD SP, NNNN	31 NN NN	10
LD D	ED A8	16
LDI	ED A0	16
LDDR	ED B8	21/16
LOIR	ED B0	21/16
NEG	ED 44	8
NOP	00	4
OR (HL)	B6	7
OR (IX + dis)	DD B6 XX	19
OR (IY + dis)	FD B6 XX	19
OR A	B7	4
OR B	B0	4
OR C	B1	4
OR D	B2	4
OR E	B3	4
OR H	B4	4
OR L	B5	4
OR NN	F6 NN	7
OTDR	ED BB	21/16
OTIR	ED B3	21/16
OUT (port), A	D3 XX	11
OUT (C), A	ED 79	12
OUT (C), B	ED 41	12
OUT (C), C	ED 49	12
OUT (C), D	ED 51	12
OUT (C), E	ED 59	12
OUT (C), H	ED 61	12
OUT (C), L	ED 69	12
OUTD	ED AB	16
OUTI	ED A3	16
POP AF	F1	10
POP BC	C1	10
POP DE	01	10
POP HL	E1	10
POP IX	DD E1	14
POP IY	FD E1	14
PUSH AF	F5	11
PUSH BC	C5	11
PUSH DE	D5	11
PUSH HL	E5	11
PUSH IX	DD E5	15
PUSH IY	FD E5	15
RES 0, (HL)	CB 86	15
RES 0, (IX + dis)	DD CB XX 86	23
RES 0, (IY + dis)	FD CB XX 86	23
RES 0, A	CB 87	8
RES 0, B	CB 80	8
RES 0, C	CB 81	8
RES 0, D	CB 82	8
RES 0, E	CB 83	8
RES 0, H	CB 84	8
RES 0, L	CB 85	8
RES 1, (HL)	CB 8E	15
RES 1, (IX + dis)	DD CB XX 8E	23
RES 1, (IY + dis)	FD CB XX 8E	23
RES 1, A	CB 8F	8
RES 1, B	CB 88	8
RES 1, C	CB 89	8
RES 1, D	CB 8A	8
RES 1, E	CB 8B	8
RES 1, H	CB 8C	8
RES 1, L	CB 8D	8
RES 2, (HL)	CB 96	15
RES 2, (IX + dis)	DD CB XX 96	23
RES 2, (IY + dis)	FD CB XX 96	23
RES 2, A	CB 97	8
RES 2, B	CB 90	8
RES 2, C	CB 91	8
RES 2, D	CB 92	8
RES 2, E	CB 93	8
RES 2, H RES 2, L	CB 94 CB 95	8 8
RES 3, (HL)	CB 9E	15
RES 3, (IX + dis)	DD CB XX 9E	23
RES 3, (IY + dis)	FD CB XX 9E	23
RES 3, A	CB 9F	8
RES 3, B	CB 98	8

RES 3, C	CB 99	8
RES 3, D	CB 9A	8
RES 3, E	CB 9B	8
RES 3, H	CB 9C	8
RES 3, L	CB 9D	8
RES 4, (HL)	CB A6	15
RES 4, (IX + dis)	DD CB XX A6	23
RES 4, (IY + dis)	FD CB XX A6	23
RES 4, A	CB A7	8
RES 4, B	CB A0	8
RES 4, C	CB A1	8
RES 4, D	CB A2	8
RES 4, E	CB A3	8
RES 4, H	CB A4	8
RES 4, L	CB A5	8
RES 5, (HL)	CB AE	15
RES 5, (IX + dis)	DD CB XX AE	23
RES 5, (IY + dis)	FD CB XX AE	23
RES 5, A	CB AF	8
RES 5, B	CB A8	8
RES 5, C	CB A9	8
RES 5, D	CB AA	8
RES 5, E	CB AB	8
RES 5, H	CB AC	8
RES 5, L	CB AD	8
RES 6, (HL)	CB B6	15
RES 6, (IX + dis)	DD CB XX B6	23
RES 6, (IY + dis)	FD CB XX B6	23
RES 6, A	CB B7	8
RES 6, B	CB B0	8
RES 6, C	CB B1	8
RES 6, D	CB B2	8
RES 6, E	CB B3	8
RES 6, H	CB B4	8
RES 6, L	CB B5	8
RES 7, (HL)	CB BE	15
RES 7, (IX + dis)	DD CB XX BE	23
RES 7, (IY + dis)	FD CB XX BE	23
RES 7, A	CB BF	8
RES 7, B	CB B8	8
RES 7, C	CB B9	8
RES 7, D	CB BA	8
RES 7, E	CB BB	8
RES 7, H	CB BC	8
RES 7, L	CB BD	8

RET	C9	10
RET Z	C8	11/5
RET NZ	C0	11/5
RET C	D8	11/5
RET NC	D0	11/5
RET PE	E8	11/5
RET PO	E0	11/5
RET M	F8	11/5
RET P	F0	11/5
RÉTI	ED 4D	14
RETN	ED 45	14
RL (HL)	CB 16	15
RL (IX + dis)	DD CB XX 16	23
RL (IY + dis)	FD CB XX 16	23
RLA	CB 17	8
RL B	CB 10	8
RL C	CB 11	8
RL D	CB 12	8
RL E	CB 13	8
RL H	CB 14	8
RL L	CB 15	8
RL A	17	4
RLC (HL)	CB 06	15
RLC (IX + dis)	DD CB XX 06	23
RLC (IY + disj	FD CB XX 06	23
RLC A	CB 07	8
RLC B	CB 00	8
RLC C	CB 01	8
RLC D	CB 02	8
RLC E	CB 03	8
RLC H	CB 04	8
RLC L	CB 05	8
RLC A	07	4
RLD	ED 6F	18
RR (HL)	CB 1E	15
RR (IX + dis)	DD CB XX 1E	23
RR (IY + dis)	FD CB XX 1E	23
RR A	CB 1F	8
RR B	CB 18	8
RR C	CB 19	8
RR D	CB 1A	
RR E	CB 1B	8
RR H	CB 1C	8
RR L	CB 1D	8
RR A	1F	4

RRC (HL)	CB 0E	15
RRC (IX + dis)	DD CB XX 0E	23
RRC (IY + dis)	FD CB XX 0E	23
RRC A	CB 0F	8
RRC B	CB 08	8
RRC C	CB 09	8
RRC D	CB 0A	8
RRC E	CB 0B	8
RRC H	CB 0C	8
RRC L	CB 0D	8
RRC A	0F	4
RRD	ED 67	18
RST 00	C7	11
RST 08	CF	11
RST 10	D7	11
RST 18	OF	11
RST 20	E7	11
RST 28	EF	11
RST 30	F7	11
RST 38	FF	11
SBC A, (HL)	9E	7
SBC A, (IX + dis)	DD 9E	19
SBC A, (IY + dis)	FD 9E	19
SBC A, A	9F	4
SBC A, B	98	4
SBC A, C	99	4
SBC A, D	9A	4
SBC A, E	9B	4
SBC A, H	9C	4
SBC A, L	9D	4
SBC A, NN	DE NN	7
SBC HL, BC	ED 42	15
SBC HL, DE	ED 52	15
SBC HL, HL	ED 62	15
SBC HL, SP	ED 72	15
SCF	37	4
SET 0, (HL)	CB C6	15
SET 0, (IX + dis)	DD CB XX C6	23
SET 0, (IY + dis)	FD CB XX C6	23
SET 0, A	CB C7	8
SET 0, B	CB C0	8
SET 0, C	CB C1	8
SET 0, D	CB C2	8
SET 0, E	CB C3	8
SET 0, H	CB C4	8
SET 0, L	CB C5	8
SET 1, (HL)	CB CE	15
SET 1, (IX + dis)	DD CB XX CE	23
SET 1, (IY + dis)	FD CB XX CE	23
SET 1, A	CB CF	8
SET 1, B	CB C8	8
SET 1.C	CB C9	8
SET 1,0	CB CA	8
SET 1, E	CB C8	8
SET 1, H	CB CC	8
SET 1, L	CB CD	8
SET 2, (HL)	CB D6	15
SET 2, (IX + dis)	DD CB XX D6	23
SET 2, (IY + dis)	FD CB XX D6	23
SET 2, A	CB D7	8
SET 2, B	CB D0	8
SET 2, C	CB D1	8
SET 2, D	CB D2	8
SET 2, E	CB D3	8
SET 2, H	CB D4	8
SET 2, L	CB D5	8
SET 3, (HL)	CB DE	15
SET 3, (IX + dis)	DD CB XX DE	23
SET 3, (IY + dis)	FD CB XX DE	23
SET 3, A	CB DF	8
SET 3, B	CB D8	8
SET 3, C	CB D9	8
SET 3, D	CB DA	8
SET 3, E	CB DB	8
SET 3, H	CB DC	8
SET 3, L	CB DD	8
SET 4, (HL)	CB E6	15
SET 4, (IX + dis)	DD CB XX E6	23
SET 4, (IY + dis)	FD CB XX E6	23
SET 4, A	CB E7	8
SET 4, B	CB E0	8
SET 4, C	CB E1	8
SET 4, D	CB E2	8
SET 4, E	CB E3	8
SET 4, H	CB E4	8
SET 4, L	CB E5	8
SET 5, (HL)	CB EE	15
SET 5, (IX + dis)	DD CB XX EE	23
SET 5, (IY + dis)	FD CB XX EE	23
SET 5, A	CB EF	8

SET 5, B	CB E8	8
SET 5, C	CB E9	8
SET 5, D	CB EA	8
SET 5, E	CB EB	8
SET 5, H	CB EC	8
SET 5, L	CB ED	8
SET 6, (HL)	CB F6	15
SET 6, (IX + dis)	DD CB XX F6	23
SET 6, (IY + dis)	FD CB XX F6	23
SET 6, A	CB F7	8
SET 6, B	CB F0	8
SET 6, C	CB F1	8
SET 6, D	CB F2	8
SET 6, E	CB F3	8
SET 6, H	CB F4	8
SET 6, L	CB F5	8
SET 7, (HL)	CB FE	15
SET 7, (IX+ dis)	DO CB XX FE	23
SET 7, (IY+ dis)	FD CB XX FE	23
SET 7, A	CB FF	8
SET 7, B	CB F8	8
SET 7, C	CB F9	8
SET 7,0	CB FA	8
SET 7, E	CB FB	8
SET 7, H	CB FC	8
SET 7, L	CB FD	8
SLA (HL)	CB 26	15
SLA (IX+ dis)	DD CB XX 26	23
SLA (IY+ dis)	FD CB XX 26	23
SLA A	CB 27	8
SLA B	CB 20	8
SLA C	CB 21	8
SLA D	CB 22	8
SLA E	CB 23	8
SLA H	CB 24	8
SLA L	CB 25	8
SRA (HL)	CB 2E	15
SRA (IX + dis)	DD CB XX 2E	23
SRA (IY + dis)	FD CB XX 2E	23

SRA A	CB 2F	8
SRA B	CB 28	8
SRA C	CB 29	8
SRA D	CB 2A	8
SRA E	CB 2B	8
SRA H	CB 2C	8
SRA L	CB 2D	8
SRL (HL)	CB 3E	15
SRL (IX + dis)	DD CB XX 3E	23
SRL (IY + dis)	FD CB XX 3E	23
SRL A	CB 3F	8
SRL B	CB 38	8
SRL C	CB 39	8
SRL D	CB 3A	8
SRL E	CB 3B	8
SRL H	CB 3C	8
SRL L	CB 3D	8
SUB (HL)	96	7
SUB (IX + dis)	DD 96 XX	19
SUB (IY + dis)	FD 96 XX	19
SUB A	97	4
SUB B	90	4
SUB C	91	4
SUB D	92	4
SUB E	93	4
SUB H	94	4
SUB L	95	4
SUB NN	D6 NN	7
XOR (HL)	AE	7
XOR (IX + dis)	DD AE XX	19
XOR (IY + dis)	FD AE XX	19
XOR A	AF	4
XOR B	A8	4
XOR C	A9	4
XOR D	AA	4
XOR E	AB	4
XOR H	AC	4
XOR L	AD	4
XOR NN	EE NN	7

VLIV INSTRUKCÍ NA STAVOVÉ INDIKÁTORY Z80

Instrukce	CY	Z	P/V	S	N	H	Komentář
ADC HL, XX	*	*	v	*	0		
ADC A, X; ADD A, X	*	*	v	*	0	*	
ADD XX, XX	*	.	.	.	0	?	
AND X	0	*	P	*	0	*	
BIT b, X	.	*	?	?	0	1	Z = log. stav bitu b
CALL instrukce			bez vlivu				
CCF	*	.	.	.	0	?	
CP X	*	*	V	*	1	*	
CPD; CPI; CPDR; CPIR	.	*	V	*	1	*	P/V = 0, když BC = 0 Z = 1, když A = (HL)
CPL	.	.	.	.	1	1	
DAA	*	*	P	*	.	*	
DEC X	.	*	V	*	1	*	8-bitová operace!
DEC XX			bez vlivu				
DI; EI			bez vlivu				
DJNZ			bez vlivu				
EX;EXX instrukce			bez vlivu				
HALT			bez vlivu				
IM0; IM1; IM2			bez vlivu				
INC X	.	*	V	*	0	*	8-bitová operace!
INC XX			bez vlivu				
IN A, (port)			bez vlivu				
IN r, (C)	.	*	P	*	0	*	
IND; INI	.	*	?	?	1	?	Z=1, když B = 0
INDR; INIR	.	1	?	?	1	?	Z = 1, když B = 0
JP; JR instrukce			bez vlivu				
LD A, I; LD A, R	.	*	IFF	*	0	0	P/V zobrazí stav IFF
LDD; LDI	.	.	*	.	0	0	
LDDR; LDIR	.	.	0	.	0	0	P/V = 0, když BC = 0
ostatní LD instrukce			bez vlivu				
NEG	*	*	V	*	1	*	
NOP			bez vlivu				
OR X	0	*	P	*	0	*	
OTDR; OTIR	.	1	?	?	1	?	Z=1, když B = 0
OUTD; OUTI	.	*	?	?	1	7	Z = 1, když B = 0
OUT (C), r; OUT (port), (A)			bez vlivu				
RET instrukce			bez vlivu				

POP XX; PUSH XX			bez vlivu			
RES b, X			bez vlivu			
RLA; RLCA; RRA; RRCA	*	.	.	.	0	0
RLD; RRD	.	*	P	*	0	0
RL X; RLC X; RR X; RRC X	*	*	P	*	0	0
SLA X; SRA X; SRL X						
SBC HL, XX	*	*	V		1	?
SCF	1	.	.	.	0	0
SET b, X			bez vlivu			
SBC X; SUB X	*	*	V	*	1	*
XOR X	0	*	P	*	0	*

Vysvětlivky

0 a 1 – indikátor je ve stavu log. 0 nebo log. 1
*** – stav indikátoru je závislý na výsledku operace**
? – stav indikátoru je nepředpokládatelný
P – ovlivnění indikátoru parity
V – ovlivnění indikátoru přetečení
. – stav indikátoru se nemění
X – 8 bitů (registru nebo adresy paměti)
XX – 16 bitů (16 –bitového i párového registru nebo dvou adres)
r – 8 –bitový registr
b – pořadové čislo bitu v bajtu X
MFF – Interrupt enable flip –flop (klopný obvod přerušení)

Pozn.: Pro instrukce CPI, CPD, LDI a LDD (včetně opakování) dále platí:
– podmínka PE je splněna, když obsah BC je různý od 0
– podmínka PO je splněna tehdy, když BC = 0

LITERATURA

Barrow David:

Assembler Routines for the Z80 Century Communications, London, GB, 1985.120 univerzálních assemblerových rutin pro programování mikroprocesoru Z80, podrobný komentář, 192 stran.

Wadsworth Nat:

Z80 Instruction Handbook, Hayden Book Company, New Jersey, USA, 1978. Ucelený a systematický přehled funkcí všech instrukcí Z80 se stručnými komentáři, 118 stran.

Nichols J. C. a kol.:

Nanocomputer SGS-ATES, USA, 1978. Dvě doprovodné učebnice k mikropočítačové stavebnici. První se zabývá programováním ve strojovém kódu assembleru Z80 (290 stran), druhá výukou interfacingu (500 stran). Velmi podrobné.

Zilog Components Data Book 1983/84, USA

Firemní publikace s popisy mikroprocesorů Z80, Z800, Z8000 a příslušejících PIO, SIO, CTC, DMA aj. obvodů z výrobního programu firmy Zilog. Obsahuje soubory instrukcí, diagramy časových průběhů a řízení všech integrovaných obvodů, 850 stran.

Bishop Graham:

Spectrum Interfacing and Projects, McGraw-Hill Book Company, GB, 1983. Návody ke stavbě hardwarových doplňků k ZX Spectru, 136 stran.

Logan Ian:

Understanding Your Spectrum, Melbourne House Publishers, GB, 1982. Stručný popis mikropočítače ZX Spectrum s ukázkami využití assembleru pro ovládání jeho hardwaru, 190 stran.

Logan I., O'Hara F.:

The Complete Spectrum ROM Disassembly, Melbourne House Publishers, GB, 1983. Komentovaný výpis obsahu paměti ROM mikropočítače ZX Spectrum. Velmi vhodný doplněk ke studiu assembleru Z80, 230 stran.

L. A. Leventhal, W. Saville:

Z80 Assembly Language Subroutines, Osborne/McGraw-Hill, USA, 1983. Souhrnná učebnice programování Z80, doplněná řadou praktických rutin i pro práci s periférierniemi, 497 stran.

Godden Bruce:

CPC 464 Firmware Amsoft, GB, 1984. Zevrubný komentovaný soubor softwarového ovládání hardwaru mikropočítače Amstrad CPC 464, cca 450 stran.

Kolektiv autorů:

Výpočetní a řídící technika. Oborová encyklopedie SNTL, 1982. Encyklopedie s abecedně řazenými hesly, 372 stran.

Dědina B. Valášek P.:

Mikroprocesory a mikropočítače. Knižnice výpočetní techniky, SNTL, 1983. Stručný přehled základních pojmů, instrukcí a mikroprocesorů 8080, MC6800, Z80, 8085, 8086, jejich odvozenin a hardwarových doplňků, 304 stran.

Kolektiv autorů:

Anglicko-český a česko-anglický elektrotechnický a elektronický slovník, SNTL, 1982. Obsahuje řadu termínů z výpočetní techniky, a to i v jejich nejčastěji užívaných anglických zkratkách, 924 stran.

O. Minihofer, J. Kratochvílová:

Anglicko-český slovník výpočetní techniky, SNTL, 1986. Obdoba předchozího slovníku, ale se specifickým zaměřením na výpočetní techniku. 494 stran.

SOFTWARE——————————————————————————pro ZX Spectrum

Machine Code Tutor, Supercode 3

První z uvedených je velmi názorný program pro výuku programování mikroprocesoru Z80. Možno vytvářet i vlastní krátké rutiny - chyby jsou ošetřeny, proto nemají kolapsové následky. Supercode 3 obsahuje 122 krátkých pomocných rutin ve strojovém kódu, které je možno využít ve vlastních programech.

GENS3, MONS3, GENS3E, M0N2, MRS, LASER GENIUS

Generátory a monitory strojového kódu. GENS3E je vylepšenou verzi GENS3 (má celoobrazovkový editor, 42 znaků/ř.). MON2 byl prvním spectrovským monitorem s funkci tracing. MRS je první původní překladač/editor assembleru Z80. LASER GENIUS obsahuje překladač/editor umožňující práci s knihovnami programů a makroinstrukcemi, jeho monitor má analyzér na bázi jazyka Forth.

SOFTWARE——————————————————————————pro Amstrad/Schneider

GENA, MONA, MEGAMON, PIRADEF, LASER GENIUS

První dva jsou obdobou GENS3 a M0NS3 pro Spectrum. MEGAMON je monitor s dobrými dispozicemi, PIRADEF je kromě překladače, editoru a monitoru vybaven pomocnými rutinami pro využití operačního systému počítače a komunikuje s jeho CP/M, LASER GENIUS (viz Spectrum) je variantou pro Amstrad/Schneider.

VĚCNÝ REJSTŘÍK

INSTRUKCE

ADC A, X — 117 – 188
ADC HL, XX — 120
ADD A, X — 116, 73 – 75
ADD XX, XX — 120
AND — 97, 99, 100, 73 – 75
BIT — 104
CALL — 84 – 85, 80, 88
CCF — 96, 82
CP — 120
CPD; CPDR — 121
CPI; CPIR — 121
CPL — 9, 6, 100
DAA — 118 – 120, 123
DEC X— 52,116
DEC XX — 120
DI — 132—133
DJNZ — 84, 85, 86
EI — 132—133
EX; EXX — 70, 77 – 78
HALT — 137
IM — 135
IN — 131
INC X — 52,116
INC XX — 120
IND; INDR — 132
INI; INIR — 131
JP — 52, 55, 56, 58, 79 – 81, 83 – 84, 127
JR — 67, 85, 86, 118, 127
LD — 47 – 50, 53, 54
LD A, I; LD A, R; LD I,A — 135
LDD — 51
LDDR — 51, 57
LDI — 51, 58
LDIR — 51, 60, 90, 91
NEG — 96
NOP — 140
OR — 98
OTDR; OTIR — 132
OUTD; OUTI — 132
OUT — 132
POP; PUSH — 68 – 69, 77 – 78, 137 – 138
RES — 104
RET— 84—85, 80, 133, 136
RL — 106, 107
RLC — 105
RLD— 110
RR — 106, 107
RRC — 105
RRD — 110, 114
RST — 65, 88
SBC HL, XX — 120
SBC X — 117
SCF — 82, 123
SET — 104
SLA — 108
SRA — 109
SRL — 110
SUB — 116
X0R — 98, 100

OSTATNÍ HESLA

adresování
" bitové — 66
" bezprostřední — 48
" " rozšířené — 48
" implikované — 66
" indexované — 66
" registrové — 47
" " nepřímé — 48
" relativní — 67
" rozšířené — 49
" RST — 65
bajt/bit — 11 – 17
Booleova algebra — 29 –30
cyklus/smyčka — 40 – 41, 85
čísla
"BCD — 118 – 120, 111, 113 – 115
" binární — 25 – 28
" hexadekadická — 25 – 28
" komplementární — 62 – 65
" multibitová — 95
data vs. instrukce — 23
dělení čísel — 125
experiment (jeho obsah)
" blokové prohledávání —126
" definované bajty — 93
" dělení 16-bit. č. 8-bitovým — 125
" DJNZ — 85
" indexované adresování — 73, 74
" logické operace — 99, 100
" maskování — 100
" násobení dvou 16-bit. č. — 122
" odečítání dvou N –bajt. č. — 123
" porovnávání — 126
" přenos parametrů — 90 – 92
" přerušení — 137
" převod kódu ASCII/bin. — 113
" " " BCD/ASCII — 114
" " " bin./ASCII —112
" PUSH; POP; EX; EXX — 77
" RST — 88
" rutina/subrutina — 86
instrukce
" aritmetické — 116 – 121
" bitových manipulaci — 104
" blokové — 51, 121, 131 – 132
" logické — 95 –99
" podmíněné — 83 – 85
" porovnáváni — 120 –121
" posuvů — 108 –110
" přenosu dat — 68
" přerušeni — 132 –133
" rotační — 105 –108, 110 – 111
" vstupně/výstupní — 131 – 132
" výměn registrových bank — 70
" zastavení — 137
" změn programového řízení — 79 – 81, 83 – 85
instrukční soubor Z80 (rozdělení) — 45 – 47
interface — 129
maskování — 97
mnemonika/mnemonický kód — 31 – 33
módy adresovací — viz adresování
" přerušení — 133
násobení čísel — 122
paměť — 20 –22
paměťová mapa — 22, 89
přenos parametrů – 90 – 92
přerušení — 132 – 136
" maskovatelné — 133
" nemaskovatelné — 133 –134
registry — 33 –36
rutina/subrutina – 86
stavové indikátory — 81 – 83, 40 – 41, 116 – 117
tabulka dat — 73, 77, 93 – 94, 115, 127
vývojový diagram — 38 – 39
zásobník — 68 – 70, 78, 85

program start

K této publikaci vydá Mladá fronta ve spolupráci se ZENITCENTREM ÚV SSM rozšířenou verzi programu MEMORY RESIDENT SYSTEM (MRS) autora Ivana Jedličky. MRS se stane cenným pomocníkem každého programátora pracujícího v assembleru mikroprocesoru Z80. Jeho pomocí lze vyvíjet a ladit vlastní programy. Obsahuje úplný obrazovkový editor na přípravu zdrojových programů EDI, překladač assembleru Z80, spojovací program LNK, program na obsluhu knihovny LIB, zpětný překladač DIS a ladicí prostředek DBG, kterým lze vytvářené programy ladit. Kazeta s programem je určena pro mikropočítače ZX Spectrum, Delta a Didaktik Gama.

Nahranou kazetu s manuálem (přibližná cena 100 Kčs) si můžete koupit v těchto prodejnách:

Knihkupectví Mladá fronta, Spálená 53, Praha 1,
Kniha 11 01 51 Palackého 9, Praha 1,
Kniha 11 02 43 Karlovo nám. 19, Praha 2,
Knihkupectví Melantrich, Na Příkopě 3, Praha 1,
Středočeské knihkupectví a nakl. prodejna 01 03 43, Olbrachtova 2809, Kladno,
Kniha 02 01 43 Žižkovo nám. 25, České Budějovice,
Kniha 03 05 60 Náměstí Republiky 9, Plzeň,
Knihkupectví u Radnice, Kniha 04 05 01, Liberec,
Kniha 050243 Čs. armády 28, Hradec Králové,
Kniha 0601 43 Česká 32, Brno,
Kniha 07 06 43 Ostružnická 1, Olomouc.

Anebo vyplňte připojenou objednávku a odešlete Mladé frontě, odbytu knih, Chlumova 10, 130 00 Praha 3, odkud Vám bude kazeta s manuálem zaslána poštou na dobírku, organizacím na fakturu.

Mladá fronta, odbyt knih
Chlumova 10
13000 Praha 3

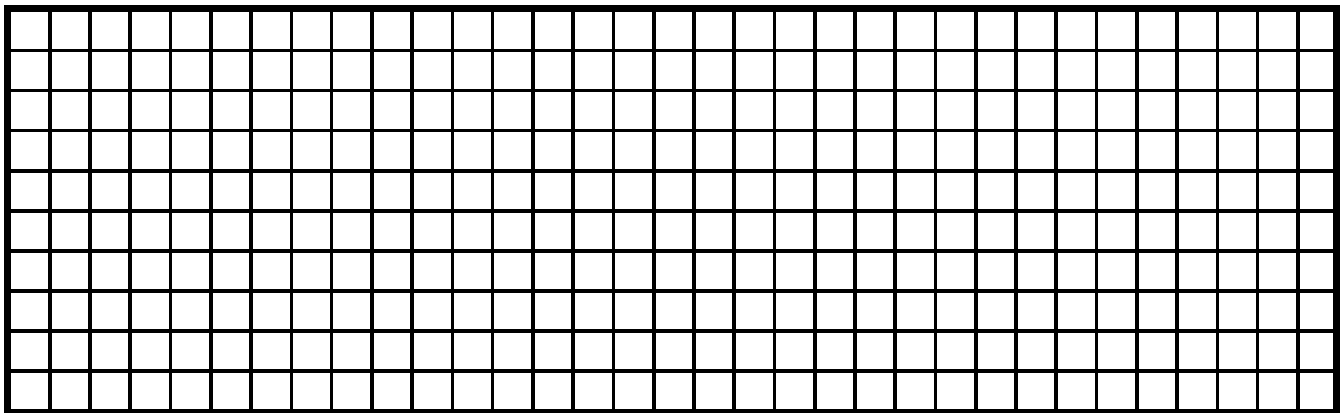
OBJEDNÁVKA

Objednávám.....kazetu (kazety) s manuálem — přílohu k publikaci Ladislava Zajíčka, Bity do bytu (program Memory Resident System)

Odešlete dobírkou na adresu

jméno a příjmení

adresa a PSČ



LADISLAV ZAJÍČEK

BITY DO BYTU

Základy programování ve strojovém kódu — assembleru Z80

Předmluvu napsal Jiří Franěk
Odbornou revizi provedl a odborně lektoroval ing Petr Adámek
Obálku navrhl a ilustroval J. Baierle
Graficky upravila Jana Vysoká
(podle osnovy J. Baierleho)
Vydala Mladá fronta jako svou 5 035 publikaci
Odpovědná redaktorka Božena Fleissigová
Výtvarný redaktor Josef Velčovský
Technická redaktorka Jana Vysoká
Z fotosazby Univers digiset zhotovené v záv. 4 n. p. Svoboda
vytiskl Mír, novinářské závody n. p. závod 3 Opletalova 3 Praha 1
1229AA 1450VA 200 stran
Náklad 25 000 výtisků 505/21/82 5
Vydáno 1 Mimo edice Praha 1988
23 059 83 03/2
Cena brožovaného výtisku 17 Kč

program